Universität Leipzig
Wirtschaftswissenschaftliche Fakultät
Institut für Wirtschaftsinformatik
Professur Softwareentwicklung für Wirtschaft und Verwaltung

Bachelorarbeit zum Thema

Anforderungen an einen Debugger für Softwaregeneratoren

Betreuender Hochschullehrer:	Prof. Dr. U. Eisenecker
Betreuender Assistent:	Max Lillack
Bearbeiter:	Christina Wagner
	Ludwig-Erhard-Str. 5
	04103 Leipzig
	Matr.-Nr.: 3679631
	6. Semester

Eingereicht am: 07.04.2016

Kurzzusammenfassung

Autor
Christina Wagner

Inhalt
Eine wichtige Aufgabe bei der Softwareentwicklung ist das Auffinden von Fehlern und das Verstehen ihrer Ursachen. Zur Unterstützung dieser Aufgabe gibt es zahlreiche Debugger. Bei der Nutzung von Softwaregeneratoren benötigt man zum Debuggen spezielle Informationen. In dieser Arbeit werden Anforderungen an einen Debugger für Softwaregeneratoren definiert. Dazu werden zunächst strukturell ähnliche Softwaregeneratoren auf ihre Debugger untersucht und grundsätzliche Debuggertypen identifiziert. Aus diesen werden 15 Anforderungen formuliert, die der hier beispielhaft betrachtete Softwaregenerator erfüllen soll. Anschließend erfolgen eine Verallgemeinerung der Ergebnisse und eine kurze Diskussion der Umsetzung auf der Plattform JetBrains MPS.

Literatur

Schlüsselwörter
Debugger, Generatoren, Anforderungsermittlung

Gliederung

Datum
07.04.2016

Gliederung

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

1.1 Motivation

Als Softwareentwickler stößt man in vielen Bereichen auf Generatoren. Bei sehr unterschiedlichen Anwendungsszenarien werden sie eingesetzt, um Codeteile automatisch zu erzeugen. Dies reicht von der Dokumentationserzeugung durch Extrahierung von Kommentaren mit JavaDoc bis zur Erstellung vollständiger Datenbankzugriffsschichten, beispielsweise mit AndroMDA. Im Rahmen des Projektes MoMaG (Modernisierung makrobasierter Generatoren) soll ADS, ein Generator für COBOL-Programme, modernisiert werden. Hierzu wurden zunächst Analysen geschrieben, die schon bestehende Makros (so heißen die Eingabedateien, die der Generator verarbeitet) auf ihre Bestandteile untersuchen und in einer XML-Datei strukturiert darstellen. Diese XML-Dateien werden dann durch einen Migrationsschritt in eine moderne Programmiersprache transformiert und gleichzeitig modernisiert. Hieraus werden im letzten Schritt mit einem ebenfalls neu erstellten Generator COBOL-Programme generiert.

Im Zuge der Entwicklung soll die Plattform, auf der ein Nutzer Makros für den Generator erstellt, ebenfalls modernisiert werden. Da große Teile der Programmiertätigkeit daraus bestehen, Fehler zu finden und zu beheben, sind Debugger ein wichtiger Bestandteil moderner Entwicklungsumgebungen. Aus diesem Grund ist ein Debugger für den modernisierten Generator geplant. Hierbei stellt sich die Frage, wie ein Debugger für einen Generator auszusehen hat und welche Ansichten und Informationen er dem Nutzer zur Verfügung stellen sollte, um ihn wirkungsvoll bei der Fehlersuche zu unterstützen. Außerdem soll ermittelt werden, wie das Verhalten des Generators dem Entwickler bestmöglich verdeutlicht werden kann. Das Ziel der folgenden Arbeit ist es, diese Fragen zu beantworten.

Der Schwerpunkt liegt dabei nicht auf der konkreten Implementierung eines Debuggers, sondern darauf, welche Informationen in welcher Weise durch den Debugger dargestellt werden, um die Fehlersuche zu unterstützen. Außerdem erfolgt die Einschränkung, dass der Debugger dem Nutzer des Generators dienen soll, also der Person, die mithilfe des Generators Programme erzeugt, nicht dem Entwickler des Generators selber. Für letzteren muss ein Debugger andere Informationen bereitstellen.

1.2 Vorgehensweise

Zur Ermittlung der Anforderungen könnte man verschiedene Wege wählen. Die Verwendung von bestehenden Generatoren und Debuggern zur Ideenfindung hat den Vorteil, dass man schon erprobte und gegebenenfalls evaluierte Ansätze nutzen kann. Außerdem bieten

dem Nutzer schon bekannte Techniken eine schnellere Eingewöhnung in die Arbeit mit dem Debugger. Deshalb ist die Untersuchung einiger Generatoren und der ihnen zugehörigen Debugger geplant. Hierbei sollen, um die Vergleichbarkeit zu gewährleisten, ebenfalls Debugger untersucht werden, die für den Nutzer eines Generators, nicht für den Entwickler, zur Verfügung gestellt werden.

Aufgrund der Fülle von Generatoren - der C-Präprozessor, SQLJ, Dokumentationswerkzeuge, Datenbankzugriffsgeneratoren, Oberflächen-Generatoren, um nur einige Beispiele zu nennen - ist es notwendig, eine Auswahl vorzunehmen. Für eine systematische und damit nachvollziehbare Auswahl werden die Generatoren zunächst nach einem Schema von Herrington [2003] klassifiziert. Anschließend erfolgt eine Auswahl von Generatoren, die ADS strukturell ähnlich sind, weil deren Debuggerfunktionalitäten am ehesten für den zu konstruierenden Debugger relevant sind. Bei den ausgewählten Generatoren wird im nächsten Schritt untersucht, ob sie einen Debugger besitzen und welche Eigenschaften und Funktionalitäten dieser aufweist. Dazu werden vorher einige Debuggertypen identifiziert und die Eigenschaften des GNU Debuggers, als ein mächtiger Vertreter der Breakpoint-Debugger, genauer beschrieben.

Im nächsten Schritt sollen alle bis dahin gewonnen Informationen ausgewertet und zusammengetragen werden, um Anforderungen an den ADS-Debugger zu formulieren. Dabei werden auch die Besonderheiten der ADS-Implementierung in MPS betrachtet, da dieses möglicherweise zur Erstellung eines Prototyps dienen wird. Eine nähere Beschreibung von ADS und MPS folgt in Kapitel 2.

Die Verallgemeinerung der Ergebnisse für Generatoren wird im letzten Schritt behandelt.

Es folgen eine Zusammenfassung der Ergebnisse, kritische Anmerkungen zur Arbeit und offene Fragen. Die Anforderungsermittlung läuft in mehreren Schritten ab, die in Abbildung 1 dargestellt sind.

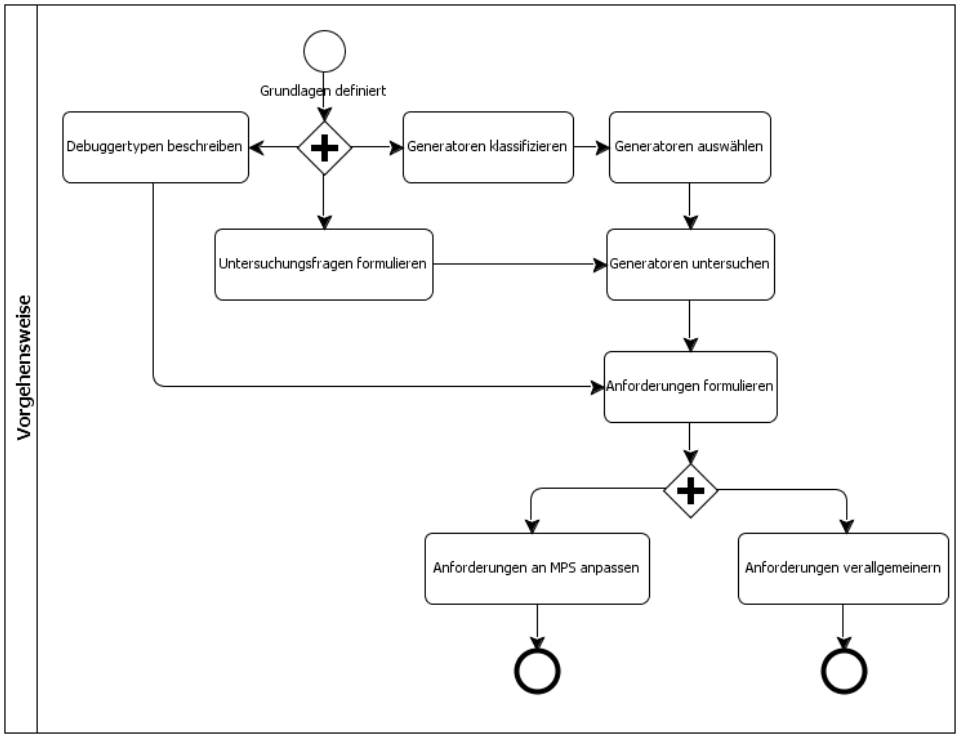

Abbildung 1: Prozess der Anforderungsermittlung (BPMN)

2 Grundlagen

2.1 Codegeneratoren

Codegeneratoren sind Programme, die andere Programme erzeugen. Die Eingabe wird von dem Benutzer des Generators entsprechend der Anforderungen erstellt und besteht aus Code in einer Programmiersprache oder aus abstrakteren Beschreibungen bis hin zu Modellen, wie einem UML-Klassenmodell. Anschließend verarbeitet der Generator diese Eingabe und erzeugt nach vorgegebenen Transformationsregeln das Generat, Quellcode, welcher anschließend kompiliert beziehungsweise interpretiert werden kann. Czarnecki und Eisenecker [2005] unterscheiden hier nach *horizontalen* und *vertikalen Transformationen*. Vertikale Transformationen, auch als *Forward Refinement* bezeichnet, implementieren und füllen schon gegebene Strukturen mit Details, verändern aber nichts an der grundsätzlichen Aufteilung der Module. Horizontale Transformationen dagegen gestalten die Struktur des Codes um, fassen Teile zusammen oder arrangieren sie komplett anders. Diese Art von Umwandlung erfordert ein komplexeres Generatorsystem (vgl. [Czarnecki/Eisenecker 2005, 341ff]). Ein Generator ist auch Teil des Compilers, der, zumeist nach vorangegangenen Schritten, aus der Hochsprache Maschinencode erstellt.

2.2 Debugger

2.2.1 Eigenschaften

Das Wort *Debugger* leitet sich vom englischen Wort *bug* ab, das ursprünglich *Käfer* bedeutet. Diese Bezeichnung für einen Programmfehler stammt angeblich von einer 1947 im Mark II gefundenen Motte, die wiederholt für Fehler im Programmablauf verantwortlich war (vgl. [Wikipedia 2016]). Mit der Zeit hat sich das Wort *bug* für einen Programmfehler im allgemeinen Sprachgebrauch durchgesetzt.

Zeller [2009, 1ff] schlägt jedoch eine genauere Unterscheidung der Begriffe vor, indem er zwischen *defect*, *infection* und *failure* unterscheidet. Hierbei beschreibt *defect* einen Codeteil, der potentiell einen Fehler verursacht. *Infection* ist der Zustand eines Programms, der vom erwarteten Zustand abweicht und durch den *defect* verursacht wird. Diese *infection* wird dann sichtbar in Form eines *failures*, wenn das entsprechende Code-Segment ausgeführt wird. Wichtig wird diese Differenzierung, wenn es zwar einen *defect* im Code gibt, dieser aber nicht als *failure* auftritt, weil die Testfälle den entsprechenden Codeteil nicht ausführen. Übertragen bedeutet dies, dass man bei der Ausführung eines Programms gelegentlich einen *failure* findet, dessen Ursprung, der *defect*, mithilfe eines Debuggers gefunden werden soll. Dies ist die erste wichtige Funktion eines Debuggers, auf die auch schon sein Name hinweist. Außerdem lässt sich mithilfe eines Debuggers ein Programm dynamisch untersuchen, auf diese zweite Funktion weist auch Rosenberg hin ([vgl. Rosenberg 1996, 1]). Zwar hängt der genaue Programmablauf von den konkreten Eingabedaten ab, trotzdem lässt sich mit repräsentativen Daten der Programmablauf verdeutlichen.

In dieser Arbeit wird der Prozess der Debugger-Nutzung und Fehlerfindung, vom Sammeln der angezeigten Informationen bis zur Bearbeitung des Quellcodes als *Debug-Prozess* bezeichnet. Alle angezeigten Informationen, die zum Zweck des Fehlerfindens und Debuggens von der Anwendung zur Verfügung gestellt werden, seien es Zeilennummern, Variablenwerte oder Fehlermeldungen, werden als *Debug-Informationen* bezeichnet. Eine genauere Darstellung von Debuggertypen, die zum besseren Verständnis dient, findet sich in Abschnitt 3.1.

2.2.2 Vier Prinzipien nach Rosenberg

Rosenberg [1996, 7ff] nennt vier wichtige Prinzipien, die ein Debugger erfüllen sollte.

Das *Heisenberg-Prinzip* (vgl. [Rosenberg 1996, 7]) besagt, dass ein Debugger das von ihm beobachtete Programm und dessen Ablauf nicht beeinflussen darf. Dieses Prinzip kann laut Rosenberg nicht komplett eingehalten werden, denn um ein Programm zu beobachten, muss

man das Betriebssystem, auf dem es läuft, nutzen und dies stellt schon eine mögliche Beeinträchtigung dar (vgl. [Rosenberg 1996, 7f]). Allerdings sollte ein Debugger dieses Prinzip so weit erfüllen, dass sich das Programmverhalten nicht ändert, während das Programm untersucht wird.

Truthful debugging lautet das zweite Prinzip, welches besagt, dass ein guter Debugger immer den tatsächlichen Programmzustand abbildet, ein Nutzer sollte sich auf den Debugger verlassen können. Dies ist zum Beispiel mit einer größeren Herausforderung verbunden, wenn der Compiler zur Übersetzungszeit Optimierungen vornimmt, sodass eine Variable nicht mehr an ihrem alten Speicherplatz zu finden ist. Wenn ein Debugger falsche Informationen anzeigt, kann das zu langem Fehlersuchen an der falschen Stelle führen.

Das dritte Prinzip wird von Rosenberg *Context is the Torch in a Dark Cave* genannt. Er nennt alle Informationen, die während eines Programmlaufs angezeigt werden, Kontext. Hierzu zählen die aktuelle Quellcode-Zeile, der Aufrufstapel, Variablenwerte, der aktuelle Thread und eine Sicht auf die Betriebssystem- und Hardwareressourcen. Diese Informationen dem Nutzer bereitzustellen, sieht Rosenberg als essentielle Eigenschaft eines Debuggers an ([vgl. Rosenberg 1996, 9ff]).

Das letzte Prinzip nennt sich *Debugging Trails System Development* und beschreibt weniger eine Anforderung an Debugger als vielmehr die Feststellung, dass bei vielen neu entwickelten Systemen zunächst keine oder nur eine sehr einfache Debugger-Unterstützung vorhanden ist. Die Debugger-Entwicklung hinkt oft den aktuellen Anwendungen hinterher. Dies ist aber als Aufforderung gedacht, zugleich mit einer neuen Anwendung auch die passende Debugger-Unterstützung auszuliefern.

2.3 Der COBOL-Generator ADS

ADS steht für *Application Development System* und ist ein kostenpflichtiger, proprietärer Softwaregenerator des Unternehmens Delta Software Technology GmbH. Technisch gesehen läuft der Generator in der PDL Generator Engine, einer virtuellen Maschine, die den ADS-Quellcode zu sogenanntem Microcode kompiliert und anschließend interpretiert. Eine Generator-Eingabe nennt sich Makro. Dieses Makro setzt sich aus COBOL-Anweisungen, ADS-Anweisungen und Prozessorcode zusammen. ADS wird im mitgelieferten Handbuch [Delta Software Technology GmbH 2012, 15] als nach dem strukturierten Paradigma arbeitend bezeichnet. Dies lässt sich auch leicht bestätigen: Es gibt Sequenzen, Auswahl und Wiederholung, während die Anweisung *GOTO* fehlt und durch Unterprogramme eine gut strukturierte Aufteilung und Wiederverwendbarkeit der Komponenten gewährleistet werden kann.

Prozessoren nennt man die 18 Untergeneratoren, aus denen ADS besteht. Diese Untergeneratoren erfüllen jeweils eine eigene Funktion, es gibt beispielsweise Prozessoren, um ein Programmgerüst zu generieren, um Programmteile strukturiert zu programmieren, für die Dokumentation oder auch für die Datenbank-Integration. Zudem hat jeder Prozessor eine eigene Syntax und verschiedene zulässige Sprachmittel.
Die Ausgabe des Generators ist ein vollständiges COBOL-Programm. Der wesentliche Vorteil der Verwendung von ADS gegenüber der Programmierung von COBOL-Anweisungen per Hand sind die mitgelieferten Delta-Standardmakros. Diese gibt es für viele standardisierte Aufgaben und sie können auch von den Prozessoren genutzt werden. Außerdem hilft der Generator, durch verfügbare COBOL-Programmgerüste, die erzeugten Anwendungen sinnvoll und gut lesbar aufzubauen.
ADS besitzt für seine derzeitige Implementierung einen Debugger, den sogenannten *Post-Generation-Debugger* (PGD). Dieser funktioniert so, dass man erst nach der Generierung das Programm inspizieren kann. Eine nähere Betrachtung dieses Werkzeuges findet sich in Kapitel 3.3.3.

Als nächstes folgt eine Beschreibung einiger wichtiger Sprachmittel von ADS, auf die im Laufe der Arbeit wiederholt eingegangen wird. Ein grundlegendes Konzept ist das Locations-Konzept. Locations beschreiben Abschnitte im Generat, die dieses unterteilen. Der Prozessor PROG, welcher ein Programmtemplate zur Verfügung stellt, legt standardmäßig schon einige Locations an, es können jedoch auch eigene definiert werden. Mit der Anweisung *.SL* können diese Locations dann jederzeit wieder aufgerufen und weitere COBOL-Anweisungen an ihr Ende generiert werden. Dies ist beispielsweise nützlich, wenn eine COBOL-Anweisung nur bedingt generiert wird und hierfür benötigte Variablen auch im Nachhinein noch in den Deklarationsteil des Programms geschrieben werden müssen.
Für die Strukturierung der Eingabedateien ist die Aufteilung in ein Startmakro und beliebig viele Untermakros von großer Bedeutung. In diese Untermakros können beispielsweise Variableninitialisierungen, bestimmte Funktionalitäten eines Programms und vieles mehr ausgelagert und mit der Anweisung *.ADD* aufgerufen werden. Da ADS keine prozedurale Programmiersprache ist, gibt es wenig optische Strukturierung und keine einfache Wiederverwendbarkeit durch Funktionen, weshalb diese Möglichkeit ähnliche Zwecke erfüllt.
Eine weitere Besonderheit sind Entscheidungstabellen, die Teil der ADS-Anweisungen sind. Darüber hinaus besitzt ADS Auswahlanweisungen, Schleifen, verschiedene Konstrukte zur Zuweisung und einiges mehr.

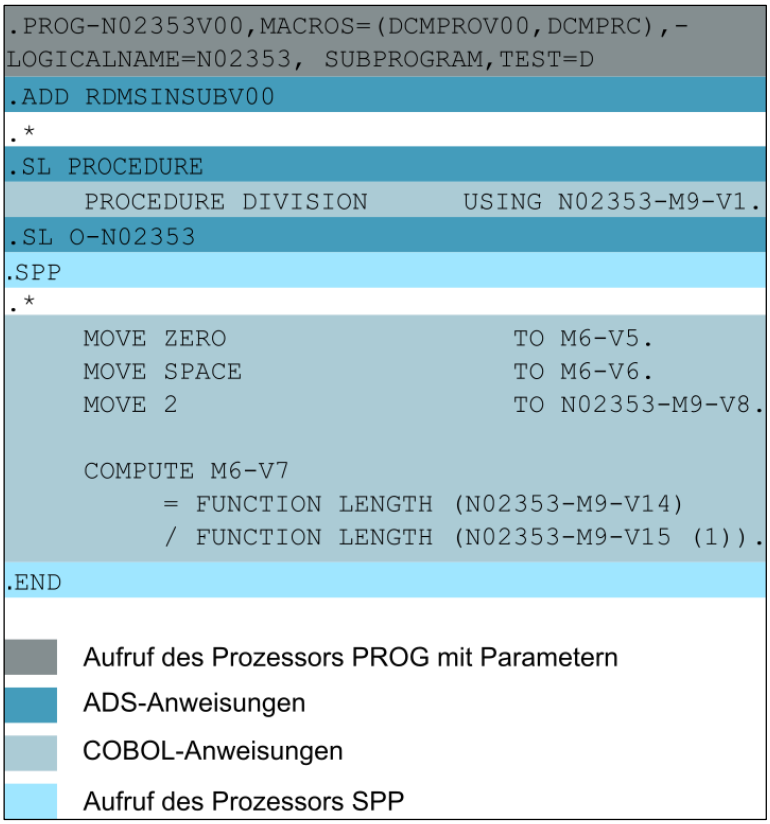

Abbildung 2: Beispiel eines ADS-Makros

2.4 JetBrains MPS

Da es im Projekt MoMaG derzeit eine Teil-Implementierung von ADS auf der Basis von Jetbrains MPS[1] gibt und diese voraussichtlich als Plattform für die Debugger-Entwicklung dienen soll, wird an dieser Stelle MPS kurz erläutert.

MPS steht für Meta Programming System, eine Plattform, die dazu dient, Programmiersprachen zu erfinden oder zu erweitern, ohne Parser oder Ähnliches erstellen zu müssen. Hierzu legt man für jeden Sprachbestandteil ein sogenanntes *Konzept* an. MPS funktioniert mithilfe eines projektionalen Editors, der das Programm intern als Syntaxbaum speichert. Die Ansicht, die dem Entwickler dabei angezeigt wird, ist lediglich eine Abbildung dieses Syntaxbaumes, die man dadurch den eigenen Bedürfnissen anpassen kann. Beispielsweise ist eine Darstellung als Pseudocode oder Tabelle möglich, was es auch Anwendern ohne Programmierkenntnisse erleichtern soll, MPS zu nutzen. Nach der Erstellung eines Programms kann aus diesem dann Code in einer anderen Sprache, zum Beispiel Java, generiert werden. Zu diesem Zweck werden entsprechende Regeln und Transformationen definiert, die bei der Generierung auf den Ursprungscode angewendet werden.

[1] https://www.jetbrains.com/mps/, abgerufen am 17.03.2016

Kehrer et al. [2013] beschreiben in einem Aufsatz die Konstruktion von Debuggern in MPS für Programmiersprachen, die um zusätzliche Sprachmittel ergänzt wurden. Um die neuen Konstrukte ebenfalls debuggen zu können, bilden sie diese auf schon bestehende Sprachmittel ab und nutzen dann einen normalen Debugger für die entsprechende Sprache. Dem Nutzer werden die Ergebnisse dann wieder an den neuen Konstrukten angezeigt. Um diesen Ansatz nutzen zu können, müsste ein Debugger für ADS schon existieren. Es gibt zwar den PGD, wie im letzten Abschnitt erwähnt, dieser kann jedoch aufgrund seiner Funktionsweise hier nicht genutzt werden.

3 Untersuchung der Debugger

Nachdem die grundlegende Vorgehensweise in Kapitel 1 und die Grundlagen dieser Arbeit in Kapitel 2 erklärt worden sind, werden im Folgenden Informationen zusammengetragen, um später Anforderungen an den Debugger zu formulieren. Dazu werden zunächst verschiedene Debuggertypen genannt und anschließend die zu untersuchenden Generatoren bestimmt. Als letzter Teil dieses Kapitels folgen die Untersuchungsfragen und -ergebnisse.

3.1 Verschiedene Debuggertypen

Um die Debugger bei der späteren Analyse strukturiert einordnen zu können und um Anregungen für die Anforderungen an den ADS-Debugger zu sammeln, folgt hier eine grundlegende Beschreibung von Debuggertypen und -funktionalitäten. Die hier dargestellten Debuggertypen orientieren sich zunächst an der Auswahl von Corley [2013]. Dieser zieht für seine Forschung zum Thema *Debugging for Model Transformations* drei Debuggertypen zur Untersuchung heran: den *Breakpoint-Debugger*, den *Omniscient-* oder *Backwards-In-Time-Debugger* und den *Query-based-Debugger* (vgl. [Corley 2013]). Als Beispiel für einen Query-based-Debugger wird in dieser Arbeit der *Whyline-Debugger* von Ko und Myers genutzt, ihn nennt Corley einen „*noteworthy query-based debugger*“ [Corley 2013]. Zusätzlich wird in dieser Arbeit *Logging* als Debuggertyp aufgeführt, weil es von Zeller als Werkzeug zum Fehlersuchen genannt wird (vgl. [Zeller 2009, 200]). Die *Transient Models* werden als Debuggertyp an dieser Stelle genannt, weil die Plattform JetBrains MPS, auf der gegebenenfalls ein Prototyp implementiert wird, diesen Typ von Debugger schon zur Verfügung stellt und sie deshalb für die Untersuchung relevant ist.

3.1.1 Logging

Zeller beschreibt Logging als die Art des Debuggings, die am wenigsten Aufwand erfordert: „*The simplest way of doing so is to have the program output the facts as desired—for instance, by inserting appropriate logging statements in the code*.“ [Zeller 2009, 200] Hierzu werden Programmzustände auf die Konsole oder in Dateien ausgegeben, um das Ablaufverhalten zu beobachten. Wenn diese Anweisungen jedoch direkt in den Quellcode eingefügt werden, führt dies zu schlecht lesbarem Code und beeinflusst gegebenenfalls auch das Programmverhalten ([vgl. Zeller 2009, 201]). Deshalb gibt es für diese Art des Debuggings Werkzeuge, beispielsweise das Programm *Log4J*. Allerdings bezeichnet Zeller [2009] Logging-Anweisungen nicht als Debugger. Ein Debugger ist nach seiner Definition durch seine Trennung vom eigentlichen Programm gekennzeichnet (vgl. [Zeller 2009, 215]). In dieser Arbeit wird eine abweichende Definition verwendet, auch eine Logging-Funktionalität wird als Debugger betrachtet.

3.1.2 Query-based-Debugger

Whyline, wie er von Ko und Myers [2008] beschrieben wird, ist ein Typ von Debugger, der vor allem auf die Nutzerfreundlichkeit zielt. Nach einem Prozess, in dem der Debugger Informationen über die Anwendung sammelt, kann der Debugger-Nutzer Fragen, die mit *Why* oder *Why didn't* beginnen, stellen. Diese werden ihm vom Debugger vorgeschlagen. Nach dem Auswählen einer Frage, zeigt der Debugger zahlreiche Informationen optisch aufbereitet an, die für das Zustandekommen des erfragten Zustandes verantwortlich sein könnten. Hier gibt es jedoch Schwierigkeiten, die Vollständigkeit der angezeigten Fragen zu evaluieren (vgl. [Ko/Myers 2008, 308]) und auch bezüglich der Zeit, in der zu Beginn Informationen gesammelt werden, sollen noch Optimierungen vorgenommen werden (vgl. [Ko/Myers 2008, 307f]).

3.1.3 Omniscient Debugger

Denker et al. [2006] beschreiben in ihrer Publikation eine Art von Debugger, mit der man ein Programm zielgerichtet rückwärts untersuchen kann. Auch der GNU Debugger enthält eine Funktionalität, mit der man im Programmablauf auf gewissen Plattformen rückwärts springen kann, jedoch ist dies nur Zeile für Zeile möglich. Der Backwards-In-Time-Debugger von Denker et al. dagegen bietet weiterreichende Funktionalitäten, wie das Markieren von Methoden und Objekten, um diese beim Rückwärtslaufen des Quellcodes verfolgen zu können oder das Formulieren von Anfragen, die anhand des Quellcodes beantwortet werden. Zur Zeit der Veröffentlichung waren noch einige Verbesserungen notwendig, um diese Art von Debugger produktiv nutzen zu können, beispielsweise bei der Ausführungsgeschwindigkeit (vgl. [Denker et al. 2006, 15]).

3.1.4 Transient Models

Die Generierung in JetBrains MPS erfolgt auf Basis von Modelltransformationen. Jedes Programm besteht aus vorher definierten Sprachbestandteilen. Für diese Bestandteile können Transformationsregeln festgelegt und während der Generierung ausgeführt werden. Jede Regel kann als ein Transformationsschritt angesehen werden, sodass bei jedem Schritt ein leicht modifiziertes Modell entsteht. Diese Modelle (sogenannte *Transient Models*) kann man während der Generierung speichern und anschließend zum Debuggen benutzen. Auf diese Weise kann der Nutzer die Reihenfolge und Auswirkungen der Transformationen nachvollziehen. Transient Models sind für Generatoren besonders geeignet, um große Transformationsschritte zu verdeutlichen.

3.1.5 Breakpoint-Debugger

Der nächste, komplexere Debuggertyp ist der wohl am weitesten verbreitete Debugger, der während der Programmausführung hilft, das Programm zu inspizieren. Hierbei erfolgt in der Regel eine symbolische Ausführung des Quellcodes, das heißt, der Maschinencode (bzw. Bytecode o.ä.) wird ausgeführt, während die entsprechende Zeile des Quellcodes angezeigt wird. Zeller nennt für diesen Typ den GNU Debugger als einen sehr mächtigen Vertreter (vgl. [Zeller 2009, 16]), deshalb folgt an dieser Stelle eine nähere Beschreibung seiner Funktionalitäten. Hierbei dient die Dokumentation des GNU Debuggers als Vorlage (vgl. [Pesch et al. 2015]). Der GNU Debugger wird hier ohne grafische Oberfläche betrachtet, wie es standardmäßig zur Verfügung steht.

Zunächst müssen während des Kompiliervorgangs Debug-Informationen erzeugt werden durch das Setzen des *-g*-Flags. Anschließend wird der Debugger mit dem Kommando *run* gestartet.

Ein wichtiger Bestandteil des Debuggers ist der Breakpoint, der gesetzt wird, um die Programmausführung an dieser Stelle, gegebenenfalls wenn eine bestimmte Bedingung erfüllt ist, zu stoppen und den aktuellen Zustand zu betrachten.

Einem Breakpoint sehr ähnlich ist der Watchpoint, der eine bestimmte Variable oder einen bestimmten Ausdruck beobachtet und das Programm stoppt, wenn diese gelesen oder verändert werden. Der Catchpoint stoppt das Programm, wenn ein gewisses Ereignis ausgelöst, beispielsweise eine Exception geworfen wird. Wenn der Programmdurchlauf gestoppt wurde, können Variablenwerte und der Zustand von Objekten betrachtet werden.

Mit den Kommandos *next* und *step* kann die schrittweise Ausführung des Programms forciert werden, sodass der Ablauf für den Nutzer gut nachvollziehbar ist. Hierbei stoppt *next* das Programm nur einmal pro Codezeile. Die Anweisung *continue* lässt das Programm normal weiterlaufen.

Auf bestimmten Plattformen kann ein Programm auch rückwärts abgearbeitet werden, die vorangehenden Schritte werden in der richtigen zeitlichen Abfolge rückgängig gemacht.

Der Aufrufstapel enthält alle aktuellen Funktionsaufrufe, angefangen bei der Einstiegsfunktion (beispielsweise die `main`-Funktion in C++) bis zur zuletzt aufgerufenen Funktion, in der Reihenfolge ihrer Verschachtelung. Zu jedem Funktionsaufruf werden zusätzlich die übergebenen Parameter und die lokalen Variablen gespeichert. Der Aufrufstapel ist nützlich bei einem vom Nutzer erzwungenen Programmstopp genauso wie bei einem Programmabbruch, ausgelöst durch einen Fehler. Der GNU Debugger kann auch gewisse Quellcodeteile anzeigen, die vom Nutzer spezifiziert werden. Diese Funktion entfällt jedoch, wenn ein Debugger eine graphische Oberfläche besitzt, hier kann der Nutzer jederzeit den gesamten

Quellcode einsehen. Zusätzlich ist es auch möglich, den passenden Maschinencode einzusehen.

Mit dem Kommando *print* können aktuelle Variablenwerte, Konstanten und ähnliches untersucht und außerdem Ausdrücke der entsprechenden Programmiersprache während der Programmausführung ausgewertet werden.

Zudem können Informationen zum Betriebssystem und zur Hardware abgefragt werden.

Der GNU Debugger bietet auch die Möglichkeit, den Quellcode während der Ausführung des Debuggers zu verändern, um Änderungen und ihre Auswirkungen direkt testen zu können.

Neben den oben genannten Funktionalitäten bietet der GNU Debugger noch viele weitere Funktionalitäten, wie das Setzen von Tracepoints für Echtzeitanwendungen oder das Debuggen von Multithread-Anwendungen.

3.2 Klassifizierung von Generatoren

Es gibt eine Vielzahl von Generatoren, die sich hinsichtlich ihres Aufbaus und ihrer Funktionsweise unterscheiden. Aus diesem Grund müsste ein Debugger für verschiedene Modelle auch unterschiedlich konstruiert sein, was später noch erläutert wird. Es ist also naheliegend, sich für den ADS-Debugger an Debuggern von strukturell und funktional ähnlichen Generatoren zu orientieren. Zu diesem Zweck werden nun verschiedene Generatortypen beschrieben und ADS wird in dieses Schema eingeordnet.

Des Weiteren dient die Klassifikation dazu, in Abschnitt 4.3 eine Verallgemeinerung der Ergebnisse auf andere Generatoren vorzunehmen.

Ein Klassifizierungsschema für Generatoren findet sich bereits bei Herrington [2003] und dient hier als Vorlage. Auch Czarnecki und Eisenecker [2005] nehmen eine Einordnung von Generatoren vor, allerdings ist diese gröber als die von Herrington und mit keinen konkreten Beispielen unterlegt, weshalb sie sich weniger gut für den folgenden Teil eignet. Herrington teilt Generatoren grundsätzlich in aktive und passive Generatoren ein. Passive Generatoren erstellen ein grundlegendes Programmgerüst, welches ein Entwickler anschließend erweitert und verändert. Aktive Programmgeneratoren zeichnen sich dadurch aus, dass sie vollständige Programme generieren, die nicht verändert werden sollten. Um Änderungen an diesen vorzunehmen, muss stattdessen die Eingabe-Datei angepasst werden. ADS gehört zu den aktiven Generatoren, es generiert vollständige COBOL-Programme. Dies wird auch verdeutlicht durch einen Hinweis im ersten ADS-Handbuch [Delta Software Technology GmbH 2012, 15]: „*Wichtig: Der von ADS generierte COBOL-Code wird NIE von Hand geändert*!“ [ursprünglich rot gedruckt, d.V.]. Aus diesem Grund werden im Weiteren nur

aktive Generatoren näher betrachtet. Unter diesen differenziert Herrington zwischen sechs grundsätzlichen Generatortypen, die vor allem nach der Art des Inputs, nach der Verarbeitung und nach dem Output klassifiziert werden.

Die Generatoren der entsprechenden Typen, die Herrington [2003] in seinem Buch nennt, werden alle untersucht. Dort, wo kein Beispiel hinterlegt sind, wurden entsprechende Generatoren ergänzt.

3.2.1 Der Code Munger

Generatoren des Typs Code Munger haben von einem Compiler oder Interpreter nutzbaren Quellcode als Input, welcher geparst wird, und aus gewissen Teilen entsteht anschließend Code in einer neuen Datei. Ein Beispiel, welches Herrington nennt, ist JavaDoc, ein Programm, welches Java-Code parst, Kommentarzeilen extrahiert und daraus eine HTML-Datei erstellt.

Zwar hat ADS keinen direkt ausführbaren Code als Ursprungsdatei, jedoch gleicht es diesem Modell insofern, dass manche Teile der Input-Datei (die COBOL-Zeilen) direkt in die Output-Datei generiert werden.

Deshalb werden einige Generatoren dieses Typs untersucht, angelehnt an Herrington sind dies Doxygen, Synopsis, JavaDoc, Scandoc, PhpDocumentor, RDoc und XDoclet. Ein Modell der Funktionsweise dieses Generatortyps und die Übertragung auf ADS finden sich in Abbildung 3.

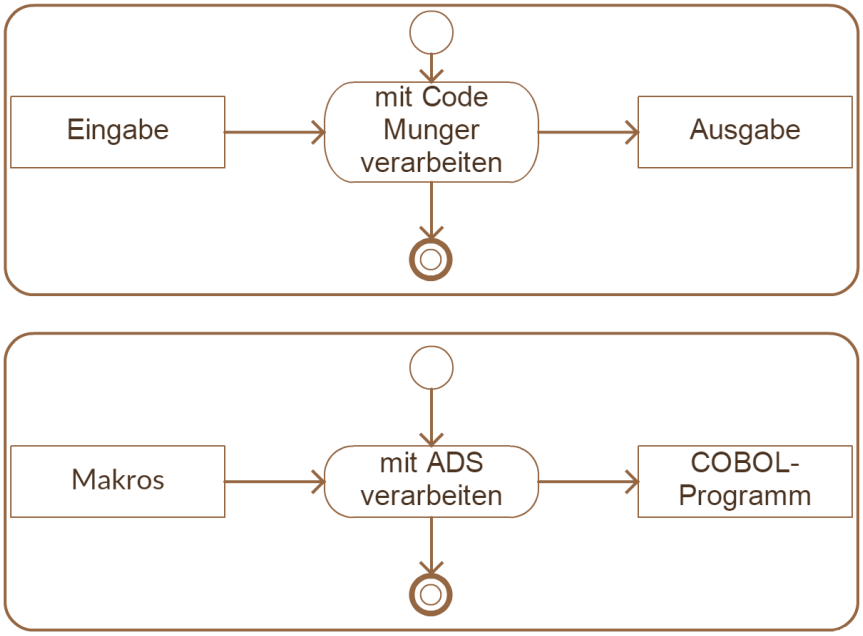

Abbildung 3: Modell des Code Munger und Übertragung auf ADS (Aktivitätsdiagramm UML 2.0)

3.2.2 Der Inline Code Expander

Ein Generator vom Typ *Inline Code Expander* dient dem Zweck komplizierte oder sehr umfangreiche Code-Teile, die eine bestimmte Funktionalität bereitstellen, durch abstraktere Definitionen im Quellcode zu beschreiben. Diese Teile werden vom Generator anschließend durch entsprechenden dem Compiler oder Interpreter verständlichen Code ersetzt. Hierbei ist wichtig, dass die Input-Datei aufgrund der Spracherweiterungen noch nicht mit einem Compiler oder Interpreter nutzbar ist. Daraus ergibt sich, dass ein Standard-Debugger der Input-Sprache nicht eingesetzt werden kann, es muss der generierte Code debuggt werden oder der Debugger muss Erweiterungen für die Generatoreingaben implementieren (vgl. [Herrington 2003, 79]).

Als wichtiges Beispiel dieses Generatortyps nennt der Autor Programme, die Embedded SQL verarbeiten. Bei diesen kann man SQL-Anweisungen nach einem Schlüsselwort direkt in den Code schreiben. Diese Anweisungen werden vom Generator erkannt und durch entsprechende Funktionen und Klassen der Programmiersprache ersetzt, mitsamt Anweisungen zum Starten und Aufrufen der Datenbank.

Da hier ein Debugger für den Generierungsprozess entworfen werden soll, kann der Generierungsprozess als Ausführung der Makros betrachtet werden. Insofern entspricht es also nicht den Generatoren vom Typ Inline-Code-Expander.

3.2.3 Der Mixed-Code-Generator

Dieser Generatortyp ähnelt dem Inline Code Expander, jedoch ist hier schon die Input-Datei ausführbar. Die speziellen Konstrukte, die der Generator später verarbeitet, werden in einer solchen Form geschrieben (beispielsweise als Kommentare der Eingabesprache), dass sie kompiliert oder interpretiert werden können und somit auch ein klassischer Debugger der Eingabesprache nutzbar ist. Außerdem wird der erzeugte Code wieder in die Input-Datei eingefügt.

Zu diesem Generatortyp sind bei ADS keine wichtigen Parallelen erkennbar, deshalb wird er nicht näher untersucht.

3.2.4 Der Partial-Class-Generator

Beim Partial-Class-Generator ist die Eingabe zweigeteilt und besteht zum einen aus einer spezifischen und vom Quellcode abstrahierten Beschreibung des Generats und zum anderen aus allgemeinen Templates für dieses. Jedoch werden hier im Gegensatz zum nächsten Modell zusätzlich handgeschriebene Codefragmente benötigt.

Auch bei ADS gibt es Templates für COBOL-Programme, diese werden durch den Prozessor PROG erzeugt. Die zu füllenden Leerstellen werden durch das Locations-Konzept repräsentiert. Allerdings ist es nicht verpflichtend, dieses Template zu nutzen, man könnte die grobe Programmstruktur auch von Hand erstellen. Bezüglich der abstrakten Beschreibung des Codes ist hier keine Nähe zu ADS erkennbar, jedoch kann dieser Aspekt durch die Konzentration auf die Templates vernachlässigt werden. Da jedoch komplette Programme und nicht nur Teile desselben generiert werden, ist die Nähe zum folgenden Generatortyp größer.

3.2.5 Der Tier Generator

Dieser Generatortyp funktioniert analog zum Partial-Class-Generator, erzeugt jedoch alle Codeteile automatisch. Damit kann eine komplette Architekturschicht (im Sinne der n-Tier-Architektur) generiert werden, woraus sich der Name ableitet. Im letzten Abschnitt ist die Ähnlichkeit zu diesem Generatortyp bereits gezeigt worden. Aufgrund dieser Ähnlichkeit werden die Tier-Generatoren untersucht, die Herrington nennt: C# Data Tier, AndroMDA, EJBGen, Ironspeed, Codecharge, TierDeveloper, Proc-Blaster, ER/Studio, Visual Studio (dialog editor), X-Designer und gSOAP. Ein Modell des Tier Generator findet sich in Abbildung 4 mit einer Übertragung auf ADS.

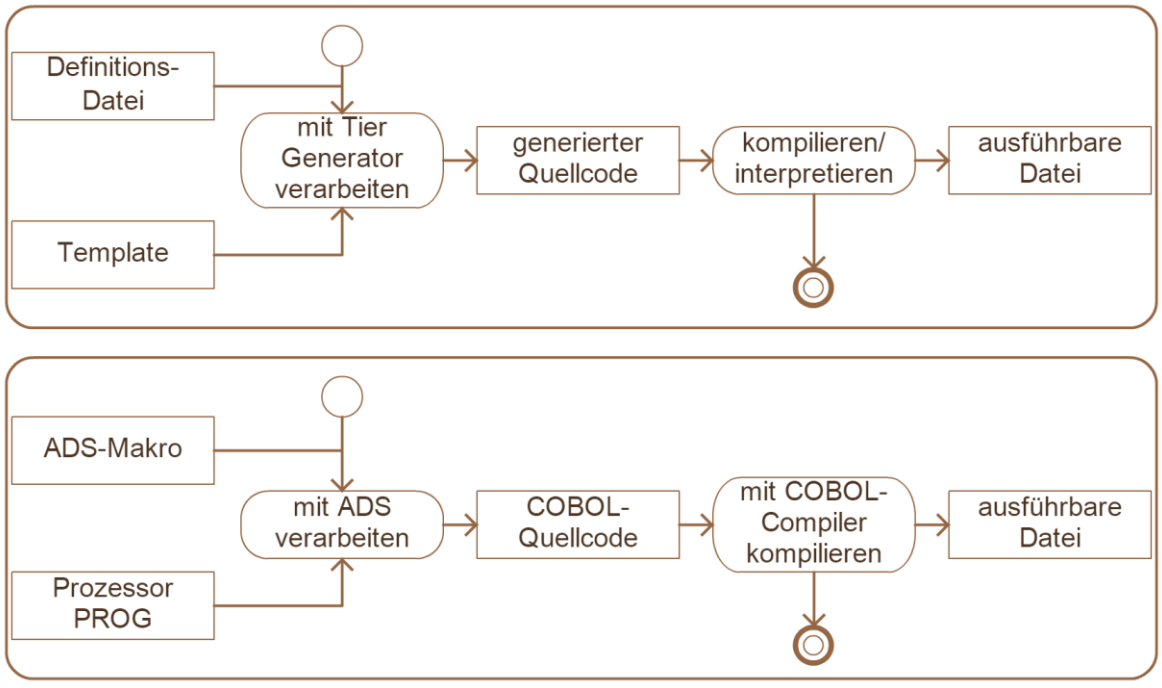

Abbildung 4: Modell des Tier Generator mit Übertragung auf ADS (Aktivitätsdiagramm UML 2.0)

3.2.6 Full Domain Language

Der sechste Generatortyp hat Turing-vollständige Sprachen als Input, aus welchen anschließend Hochsprachen generiert werden. Diese Eingabe-Sprachen orientieren sich am Einsatzgebiet der fertigen Software und sollen auch von Nicht-Programmierern dieses Fachgebiets leicht erlernbar und benutzbar sein.

Bei ADS handelt es sich zwar nicht um eine leicht erlernbare und am Einsatzgebiet der Software orientierte Sprache, jedoch ist sie Turing-vollständig, weshalb auch Generatoren dieses Typs untersucht werden.

Das Google Web Toolkit generiert aus Java-Quellcode die entsprechenden Dateien in JavaScript. JetBrains MPS und Xtext sind Language Workbenches, die eine Art Metaebene dieses Generatortyps darstellen, da man mit ihrer Hilfe Full Domain Language-Generatoren implementieren kann. Aus diesem Grund werden sie daraufhin betrachtet, welche Möglichkeiten sie dem Nutzer bieten, die selbst erstellte Sprache mit einem Debugger zu ergänzen. Bei Xtext erfolgt hier die Einschränkung, dass die Generierung mit Xtend geschieht. Xtend hat besondere Funktionalitäten, um speziell die Generierung zu unterstützen, beispielsweise Templates, sowie eine leistungsfähige Integration in die Entwicklungsumgebung Eclipse mit der Möglichkeit viele Funktionalitäten zu nutzen (vgl. [Bettini 2013]).

3.3 Untersuchung der Generatoren

3.3.1 Bestimmung der Untersuchungsaspekte

Die Untersuchung der Generatoren geschieht nach den folgenden Gesichtspunkten:

- Gibt es einen Debugger? Hiermit ist ein Debugger für den Generierungsprozess selbst gemeint, kein Debugger für das generierte Programm.
  Wenn ja, wird dieser Debugger durch die folgenden Punkte näher betrachtet:
  - Welchem Debuggertyp, wie sie in Kapitel 3.1 vorgestellt werden, kann der Debugger zugeordnet werden?
  - Welche Informationen werden angezeigt?
    Hier steht eine Liste aller vom Debugger bereitgestellten Informationen.
  - Wie wird der Debugger verwendet?
    Diese Frage zielt darauf ab, wie der Debugging-Prozess ausgelöst wird und wie der Nutzer währenddessen mit dem Debugger interagieren kann.
  - In welchem zeitlichen Zusammenhang stehen Generierung und Anzeige der Debug-Informationen?
    ADS besitzt einen sogenannten Post-Generation-Debugger, welcher erst nach der vollständigen Generierung Informationen über diese anzeigt. Dieses Verhalten

entspricht nicht der Funktionsweise der meisten aktuellen Debugger, deshalb wäre es interessant zu sehen, wie dieser Zusammenhang bei den zu untersuchenden Debuggern implementiert ist.

3.3.2 Ergebnisse der Generatoren-Untersuchung

Tabelle 1 zeigt die Ergebnisse der Generatoren-Untersuchung. Die ersten beiden Spalten geben den Generator und den Generatortyp an. Anschließend folgt die Frage, ob der Generator einen Debugger hat, und, wenn das der Fall ist, welcher Debuggertyp sich bestimmen lässt. Die Spalte *Informationen* zeigt alle vom Debugger angezeigten Informationen. Unter der Überschrift *Verwendung* findet sich die Beschreibung, wie der Debugger gestartet und benutzt wird, *Flag* bedeutet hier, dass die Generierung mit einem zusätzlichen Flag gestartet wird. Ein Beispiel hierfür ist Doxygen, bei welchem der Befehl zur Ausgabe des Markdowns, einer Debug-Ansicht, folgendermaßen lautet: `doxygen -d Markdown <config-file>`. `-d Markdown` stellt hierbei das Flag dar, `config-file` muss durch eine entsprechende Konfigurationsdatei ersetzt werden. Teilweise werden auch *Konfigurationsdateien*, zumeist als XML-Struktur übergeben, bei einer grafischen Oberfläche muss man *Checkboxen anwählen*. *Während* oder *nach* bezieht sich auf den zeitlichen Zusammenhang zwischen dem Generierungsprozess und der Ausgabe der Debug-Informationen; *während* bedeutet, dass der Debugger während der Generierung die Informationen anzeigt. Ausführliche Quellenangaben finden sich im Literaturverzeichnis. Eine umfangreiche Auswertung folgt in den nächsten Abschnitten, gegliedert nach den Generatortypen.

Generator	Generatortyp	Debugger?	Debuggertyp	Informationen	Verwendung	Zeit	Quelle
Doxygen	Code Munger	ja	Transient Model	Markdown, Kommandos	Flag	während	[van Heesch 2015]
Synopsis	Code Munger	ja	Logging	ParseTree, Tokens	Nutzung gewisser Klassen	während/nach	[Seefeld 2004]
JavaDoc	Code Munger	nein					[JavaDoc 2010]
ScanDoc	Code Munger	nein					[Talin 2000]
PhpDocumentor	Code Munger	ja	Logging	Logginginformationen	Konfigurationsdatei	während/nach	[phpDocumentor]
Rdoc	Code Munger	ja	Logging	Logginginformationen	Flag	während/nach	[RDoc 2005]
Xdoclet	Code Munger	Keine Quellen					
C# Data Tier	Tier generator	Keine Quellen					
AndroMDA	Tier generator	ja	Logging	Logginginformationen	Konfigurationsdatei	während/nach	[AndroMDA 2014]
EJBGen	Tier generator	ja	Logging	Logginginformationen	Flag	während/nach	[EJBGen Reference 2016]
Ironspeed	Tier generator	ja	Logging	Logginginformationen	Checkbox anwählen	während/nach	[Iron Speed 2015]
Codecharge	Tier generator	nein					[CodeCharge 2012]
TierDeveloper	Tier generator	Keine Quellen					
Proc-Blaster	Tier generator	Keine Quellen					
ER/Studio	Tier generator	Keine Quellen					
Visual Studio	Tier generator	nein					
X-Designer	Tier generator	Keine Quellen					
gSOAP	Tier generator	ja	Logging	Logginginformationen	Flag	während/nach	[van Engelen 2016]
Google Web Toolkit	Full Domain Language	nein					[GWT 2014]
JetBrains MPS	Full Domain Language	ja	Transient Model	Zwischenmodelle	Modell speichern	nach	[Alshannikov 2013]
				Generation Trace Tool		nach	[Alshannikov 2013]
				weitere Informationen		nach	[Alshannikov 2013]
Xtext (mit Xtend)	Full Domain Language	ja	Logging	Logginginformationen	Konfigurationsdatei	während/nach	[Efftinge 2016]
				Anzeige des Generats	Auswahl in Debug-Prozess	nach	[Bettini 2013]
				Matching von Eingabe und Generat	extra Fenster bei Eingabe	nach	[Bettini 2013]
ADS		ja	PGD	Makro und Matching von Zeilen	PGD anwählen	nach	
				Variable, Alias, Änderung	extra Ansicht		
				Aufrufgraph aller Makros			
				Aufrufgraph mit Parametern			

Tabelle 1: Ergebnisse der Generatoren-Untersuchung

3.3.3 Der ADS-Post-Generation-Debugger

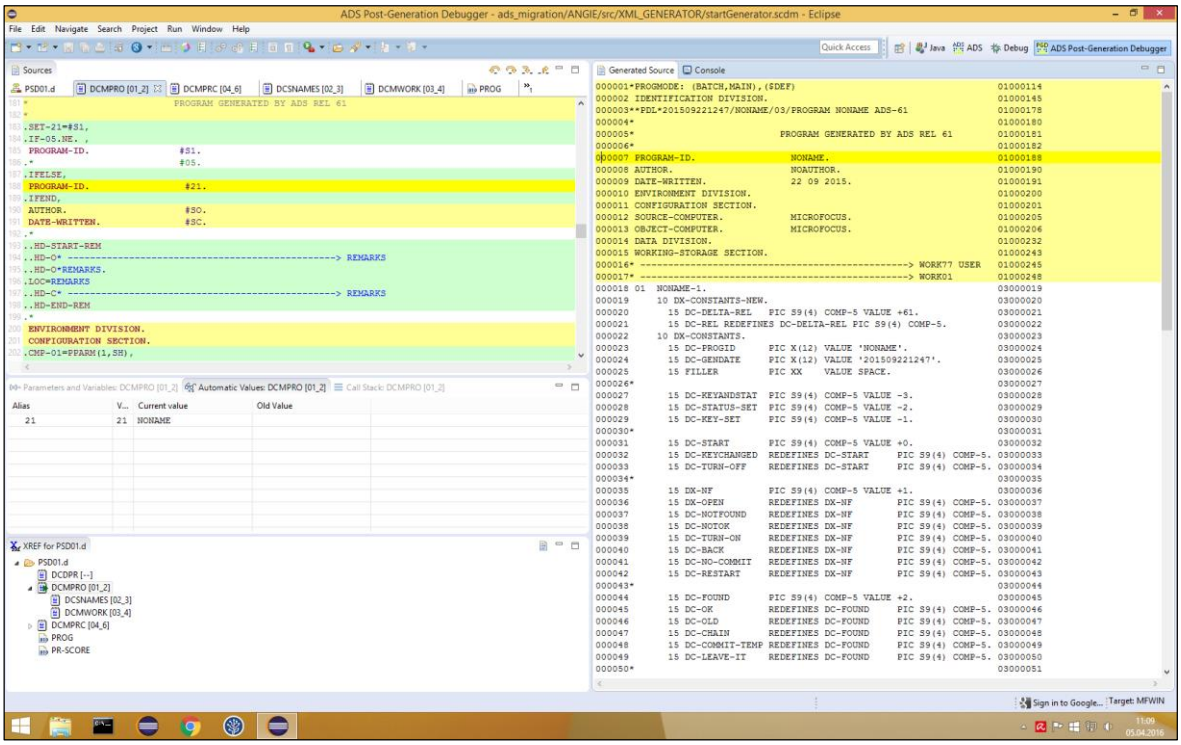

Abbildung 5: Screenshot des ADS-PGD

Der ADS-Post-Generation-Debugger (PGD) ist der Debugger der bestehenden ADS-Implementierung, Abbildung 5 stellt ihn anhand eines Screenshots dar. Er kann keinem vorher beschriebenen Debuggertyp direkt zugeordnet werden. Die Funktionsweise des PGD wird schon im Namen angedeutet: Während der Generierung des COBOL-Programms können optional Informationen für den Debugger gesammelt und anschließend in einer separaten Ansicht angezeigt werden. Dazu bietet er vier Fenster: Ein großes zeigt den generierten COBOL-Code mit zeilenweisem Herkunftsnachweis an. Dem gegenüber befindet sich ein Fenster, das die entsprechenden Makros anzeigt. In beiden Fenstern kann man einzelne Zeilen markieren, wodurch im jeweils anderen die entsprechende Zeile hervorgehoben wird, aus der der Code stammt, beziehungsweise, wohin er generiert wurde.

Ein drittes Fenster bietet dem Nutzer eine Variablensicht an, in welchem alle globalen und lokalen Variablen mit Alias und altem und neuem Wert dargestellt werden. Hier kann man auch Schleifen Schritt für Schritt durchlaufen und ihre Werte betrachten. Im letzten Fenster befindet sich ein Aufrufgraph der Makros.

Zudem gibt es vier Knöpfe, mit denen man Schritt für Schritt, vorwärts und rückwärts, durch die ausgeführten Codezeilen navigieren und in Makros hinein- und hinausspringen kann.

Der Post-Generation-Debugger unterscheidet sich insofern vom Breakpoint-Debugger und vom Omniscient Debugger, dass der Quellcode nicht automatisch durchlaufen, sondern von

Hand Zeile für Zeile beobachtet wird. Ein Vorteil hiervon besteht darin, dass man den Code sehr frei betrachten kann, ohne technische Einschränkungen, der reale Programmablauf ist jedoch Schritt für Schritt und bei großen Programmen nur sehr langsam nachzuvollziehen. Dadurch bietet der Debugger nicht in vollem Umfang die zweite Funktion, die ein Debugger nach Rosenberg erfüllen sollte (siehe Kapitel 2.2.1), nämlich die Verdeutlichung des Programmablaufes und seiner Reihenfolge. So kann auch nur mühsam die reale Generierungsreihenfolge nachvollzogen werden, da ADS durch das Locations-Konzept die Möglichkeit eröffnet, ein Programm in einer relativ freien Reihenfolge zu generieren.

3.3.4 Generatoren des Typs Code Munger

Für den Typ *Code Munger* wurden ausschließlich Dokumentationswerkzeuge untersucht, dies waren Doxygen, Synopsis, JavaDoc, Scandoc, PhpDocumentor, RDoc und XDoclet.
Unter ihnen bilden JavaDoc, und Scandoc eine Gruppe, die keinerlei Debug-Informationen bereitstellt. Zu XDoclet konnten keine aussagekräftigen Quellen gefunden werden.
RDoc kann mit dem Flag `-debug` gestartet werden. Diese Option dient den Entwicklern von RDoc zur Ausgabe relevanter Informationen, nicht den Anwendern.
PhpDocumentor bietet die Möglichkeit, Logging-Informationen in eine Log-Datei zu schreiben. Hierzu kann der Nutzer bestimmen, wie viele Informationen er bekommen möchte, ob nur Fehlermeldungen oder auch andere ergänzende Meldungen erhalten möchte.
Der Debugger wird vor der Generierung durch Angabe von Logging-Tags eingebunden. Nach der Generierung sind die gesammelten Informationen dann in einer Datei einsehbar. Außer bei der Anfangskonfiguration kann der Nutzer aber nicht mit dem Debugger interagieren.
Synopsis bietet ebenfalls die Möglichkeit des Loggings. Hier kann der Parse-Tree, der einen Zwischenschritt der Generierung darstellt, mit Kommentaren ausgegeben werden, außerdem sind die Tokens einsehbar, in welche die Eingabe zerlegt wird.
Eine Ausnahme in dieser Generator-Gruppe bezüglich der Debugging-Unterstützung bildet Doxygen. Hier wird dem Nutzer ein Transient Model angeboten. Dieses präsentiert den Zwischenstand, nachdem die Kommentare aus dem Quelltext extrahiert und mit Markdown und Kommandos versehen wurden.
Insgesamt ist die Debugging-Unterstützung bei diesem Generatortyp nur schwach ausgeprägt. Ein Grund dafür könnte sein, dass die Generatoren nur eine geringe Komplexität und wenige, einfache Anweisungen aufweisen, sodass Logging zum Fehlersuchen ausreichend erscheint.

3.3.5 Generatoren des Typs Tier-Generator

Die untersuchten Generatoren des Typs *Tier-Generator*, welche von Herrington [2003] genannt werden, sind: C# Data Tier, AndroMDA, EJBGen, Ironspeed, Codecharge, TierDeveloper, Proc-Blaster, ER/Studio, Visual Studio (dialog editor), X-Designer und gSOAP. Hier konnten zu C# Data Tier, Tier Developer, Proc-Blaster, ER/Studio und X-Designer keine aussagekräftigen Quellen gefunden werden. Sie sind zum Teil proprietäre Programme, bei denen die Dokumentation nicht frei verfügbar ist, bei Anderen wurde die Entwicklung eingestellt und es sind keine aussagekräftigen Quellen mehr aufzufinden.
Codecharge und Visual Studio bieten keine Debug-Unterstützung für den Generierungsprozess selbst an. Der Dialog Editor von Visual Studio ist jedoch ein Generator für eine grafische Oberfläche, deren Quellcode nach der Generierung debuggt werden kann.
AndroMDA, EJBGen, Ironspeed und gSOAP bieten Logging zum Debuggen der Generierung an. Die Ausgabe der Logging-Informationen kann während der Generatorkonfiguration angewählt werden.

3.3.6 Generatoren des Typs Full Domain Language

Das Google Web Toolkit, welches aus Java-Code JavaScript-Code generiert, wird als Plugin in Eclipse genutzt. Hierbei kann der vom Nutzer erstellt Java-Code mit dem Eclipse Java-Debugger untersucht werden, ebenso wie der Java-Script-Code mit entsprechenden Debuggern betrachtet werden kann. Zudem gibt es eine Abbildung des Java-Codes auf das Generat, sodass man während des Debugging-Prozesses zwischen den beiden Ansichten wechseln kann. Für den Generierungs-Prozess selbst ist jedoch kein Debugger vorhanden.
JetBrains MPS besitzt eine besondere Form des Debuggings, sogenannte Transient Models. Deren Speicherung kann vom Nutzer aktiviert werden, sodass sie nach dem Generierungs-Prozess einsehbar sind. Die Generierung in MPS verläuft unter Anwendung aufeinander folgender Transformationsschritte. Diese werden vom Nutzer definiert. Bei Nutzung der Transient Models wird nach jedem Transformationsschritt ein Modell erstellt und gespeichert, sodass für den Nutzer leichter erkennbar ist, wann ein *failure* auftritt.
Mit dem sogenannten *Generation Trace Tool* werden diese Schritte auch für eine spezielle Codezeile übersichtlich abgebildet. Zudem gibt es noch weitere kleinere Funktionalitäten, die das Debuggen unterstützen, wie die Anzeige des Typs einer Variable.
Die ADS-Implementierung in MPS nutzt für den ersten Schritt, die Verarbeitung der ADS-Anweisungen, größtenteils Java-Code. Dazu gibt es in MPS die Möglichkeit, Java-Klassen als vorbereitenden oder nachgestellten Transformationsschritt zu definieren. Dieser Schritt wird jedoch nicht in den Transient Models dargestellt. Dazu gibt es eine weitere Möglichkeit

in JetBrains MPS, die Implementierung eines generatorspezifischen Debuggers mithilfe eines Debugger-Interfaces.

Xtext ist wie das Google Web Toolkit als Plugin für Eclipse verfügbar. Wird mithilfe von Xtend Code generiert (wie hier festgelegt), gibt es die Möglichkeit während des Programmierens der Eingabe (in Xtend) das Generat zu betrachten. Es kann auch für einen speziellen Abschnitt das Generat gesondert angezeigt werden, sodass eine Art Matching von Eingabe und Generat möglich ist. Außerdem können mithilfe von Log4J selbst verfasste Statusmeldungen ausgegeben werden. Da bei jedem Speichern das Generat aktualisiert wird, und es somit keinen expliziten Generierungsschritt gibt, ist die Angabe des zeitlichen Zusammenhangs hier eigentlich hinfällig.

4 Auswertung

Im Folgenden werden alle Informationen, die bis hierhin zusammengetragen wurden, bewertet und daraus Anforderungen ermittelt.

4.1 Formulierung von Anforderungen

4.1.1 Skizze der Anforderungen

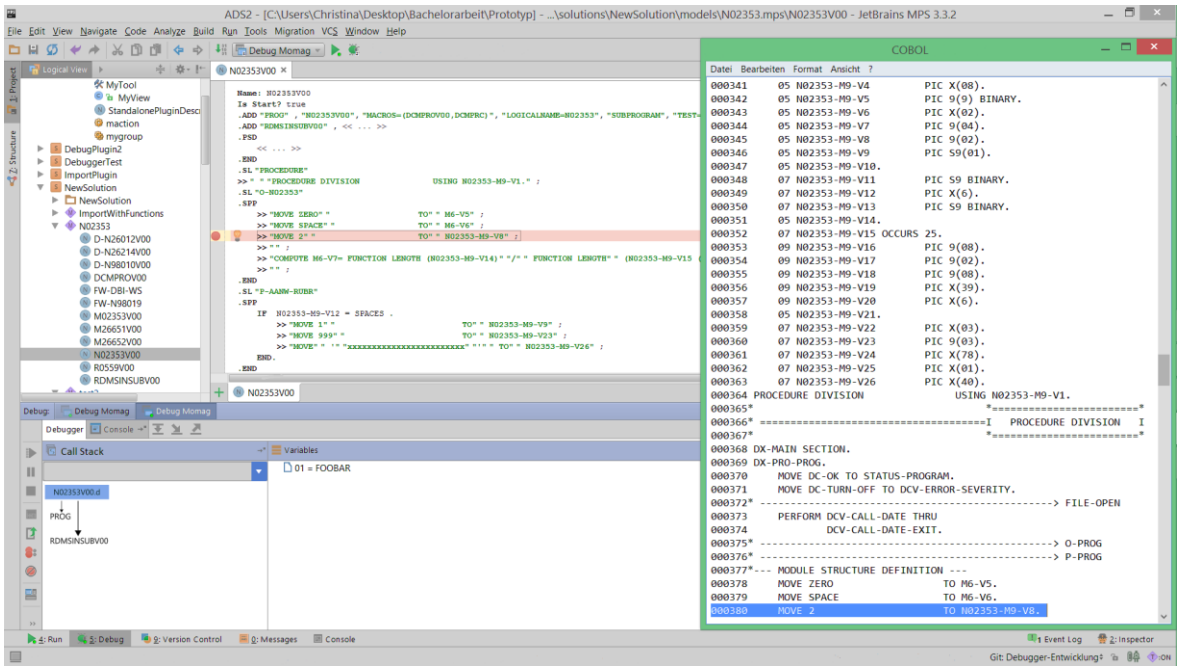

Abbildung 6: Skizze des Debuggers

In Abbildung 6 sieht man als Überblick einige der Anforderungen, die in den folgenden Abschnitten ermittelt werden, skizziert. Im mittleren Fenster ist das aktuell durchlaufene Makro zu sehen (Anforderung 5). In diesem Fenster ist eine Zeile markiert, in welcher ein Breakpoint gesetzt wurde (Anforderung 12). Rechts daneben befindet sich der bis zum Zeitpunkt des Programmstopps generierte COBOL-Code (Anforderung 5). In diesem Fenster ist die Zeile markiert, die auch im Makro markiert ist (Anforderung 5). Der untere Bereich zeigt ganz links einige Bedienelemente zum schrittweisen Durchlaufen des Makros (Anforderung 14). Rechts daneben ist beispielhaft ein Aufrufgraph der bis dahin ausgeführten Makros gezeigt (Anforderung 7). Von dort aus wiederum rechts befindet sich eine Variablenansicht, die gerade aktive Variablen und ihren Wert beinhaltet (Anforderung 6).

4.1.2 Ermittlung auf Grundlage des ADS PGD

Da derzeit schon eine Debugger-Implementierung in Form des ADS-Post-Generation-Debuggers existiert, soll es in dieser Arbeit um eine Neukonzeption des Debuggers von Grund auf gehen, gleichzeitig besteht aber der Anspruch, dass der neu entwickelte Debugger gegenüber dem Post-Generation-Debugger eine Verbesserung darstellt oder, wenn dies nicht

möglich ist, so doch keine Verschlechterung hinsichtlich der Nutzerfreundlichkeit und hinsichtlich des Nutzens für den Debug-Prozess bringt.

1. Anforderung: Der neue Debugger soll mindestens so gut bedienbar und so nützlich für den Debug-Prozess sein wie der ADS-Post-Generation-Debugger.

4.1.3 Anforderungen aus den vier Prinzipien nach Rosenberg

Die ersten drei der vier Prinzipien nach Rosenberg (wie in Abschnitt 2.2.2 beschrieben) dienen als grundlegende Anforderungen an den neuen Debugger.

2. Anforderung: Das *Heisenberg-Prinzip*, welches besagt, dass der Debugger das Programmverhalten nur minimal beeinflussen darf, soll vom neuen Debugger erfüllt werden.
3. Anforderung: Auch *Truthful Debugging* ist eine wichtige Eigenschaft für den neuen Debugger, woraus die Pflicht entsteht, diesen während und nach der Implementierung gründlich zu testen. Dieses Prinzip kann auch dadurch unterstützt werden, dass der Debugger auf MPS basiert, welches schon eine Debugger-API zur Verfügung stellt, sodass der Debugger nicht direkt auf der Betriebssystemebene aufsetzen muss. Dies vermeidet viele Fehler beim Auslesen des Speichers, die zum Beispiel durch Optimierungen des Betriebssystems entstehen.
4. Anforderung: Das dritte Prinzip *Context is the Torch in a Dark Cave* soll ebenfalls umgesetzt werden. Als Konsequenz folgt daraus, dass dem Nutzer des Debuggers möglichst viele relevante Kontextinformationen zur Verfügung gestellt werden. Welche Informationen hierfür im Einzelnen notwendig sind, wird nachfolgend näher bestimmt.

Da das vierte Prinzip, *Debugging Trails System Development*, also die Feststellung, dass oftmals moderne Softwaresysteme keinen modernen Debugger besitzen, keine Anforderung an Debugger darstellt, resultiert hieraus keine Anforderung an den neuen Debugger.

4.1.4 Anforderungen aus dem ADS-Post-Generation-Debugger

Der Post-Generation-Debugger (PGD) ist für einen dem modernisierten ADS-Generator sehr ähnlichen Generator konstruiert worden. Deshalb ist es naheliegend, zunächst dessen Eigenschaften und Funktionalitäten zu betrachten und alle weiteren Anforderungen vor diesem Hintergrund zu entwickeln.

5. Anforderung: Der Debugger stellt in zwei Fenstern die Makros und den generierten COBOL-Code dar; wenn man eine Zeile anklickt, wird die entsprechende Zeile im anderen Fenster markiert.

   Eine sehr nützliche Eigenschaft des PGD ist die Gegenüberstellung von Makro und Generat. Da die Generierung nach dem Prinzip des Code Mungers funktioniert, also viele

Zeilen direkt übernommen und in eine neue Datei geschrieben werden, können die meisten generierten Zeilen direkt den Zeilen in Makros zugeordnet werden. Nur wenige Ausnahmen stammen zum Beispiel aus den allgemeinen Templates, die der Prozessor PROG zur Verfügung stellt oder werden automatisch generiert, zum Beispiel zum Anlegen von notwendigen Variablen. Das Markieren von einander entsprechenden Zeilen dient dem Zweck der Übersichtlichkeit und zum besseren Verständnis des Generats.

6. Anforderung: Der Debugger stellt eine Variablenansicht zur Verfügung, in welcher alle zum Zeitpunkt der Betrachtung aktiven Variablen, ihre Aliasse und ihre Werte dargestellt werden. Zu einem Makro sind die Parameter und die Werte, mit denen das Makro aufgerufen wurde, einsehbar.

   Die Variablenansicht als drittes Fenster ist ebenfalls notwendig, weil es dort einige komplexe Funktionalitäten gibt. Beispielsweise können Variablen alternative, aussagekräftigere Namen (Aliasse) besitzen, die zentral vergeben werden. Genauso werden globale Variablen oft in einem Makro zentral initialisiert. Nach der Ausführung dieses Makros stehen die Variablenwerte dem Nutzer zur Verfügung, allerdings ist ihr aktueller Wert manchmal schwer nachzuvollziehen. Außerdem können lokale Variablen beim Aufruf des Makros durch übergebene Parameter initialisiert werden. Hier gibt es eine Analogie zum Konzept der Prozeduren in prozeduralen Programmiersprachen, worauf Anforderung 7 näher eingeht.

7. Anforderung: Es gibt ein viertes Fenster, welches den Aufrufgraphen mit einer Verlinkung zu der Codezeile, in der der Aufruf geschieht, darstellt.

   Auch das vierte Fenster bietet nützliche Informationen. Da ADS keine Prozeduren oder Funktionen besitzt, geschieht die Strukturierung der Eingabe hauptsächlich durch die Verteilung auf verschiedene Makros. Durch einen Aufrufgraphen kann die Aufrufreihenfolge gut nachvollzogen werden, ohne alle entsprechenden Anweisungen im Quellcode zu suchen. Außerdem gibt es oft den Fall eines bedingten Makroaufrufs, dessen Auswertung durch den Aufrufgraphen ersichtlich wird. Zu diesem Zweck wäre es noch nützlich, eine Verlinkung des Makroaufrufs im Aufrufgraph zur entsprechenden Codezeile herzustellen. Diese Funktion könnte auch die Angabe der Aufrufparameter ersetzen, weil man die Parameter direkt im Makro einsehen kann.

4.1.5 Anforderungen aus den Debuggertypen

Die einfachste Art des Debuggings ist die Ausgabe von Statusmeldungen und anderen Informationen über das laufende Programm auf die Konsole oder in eine Datei (Logging).

Hierzu kann man die Möglichkeit anbieten, gewisse Schritte während der Generierung auszugeben oder Variablen zu markieren, deren Wert während der Generierung beobachtet werden sollen. Damit würde man jedoch nur versuchen, das Verhalten nachzubilden, das sich mit einem Breakpoint-Debugger komfortabler bereitstellen lässt. Zudem muss man bei einem Breakpoint-Debugger nicht im Voraus wissen, welche Variablen man gerne beobachten will, sondern kann jede Variable, die zur Zeit eines Programmstopps aktiv ist, anschauen. Ein weiterer Nachteil des Loggings ist seine Unübersichtlichkeit. Wenn viele Informationen ausgegeben werden, was bei einem sehr komplexen Programm wie ADS durchaus der Fall sein kann, ist es schwierig, die Meldungen übersichtlich darzustellen. Hierfür könnte man mehrere Fenster nutzen, jedoch würde der neue Debugger eine Vereinfachung gegenüber dem Post-Generation-Debugger darstellen. Dies verletzt die 1. Anforderung, da davon auszugehen ist, dass für ein komplexes Programm ein komplexer Debugger notwendig ist, welcher dem Nutzer viele und gut strukturierte Informationen anbietet.

8. Anforderung: COBOL-Fragmente können vor dem Debugging markiert und während des Debuggings beobachtet werden.

   Ein Debugger vom Typ *Query-based-Debugger* beantwortet Fragen, die spezifisch für eine Anwendung zur Verfügung gestellt werden. Bei ADS wären Fragen wie *Warum wird dieses Codefragment nicht generiert?* oder *Warum enthält diese Location den Code in der vorliegenden Reihenfolge?* denkbar. Entsprechende Funktionalitäten lassen sich nachbilden, indem man bestimmte COBOL-Fragmente vor dem Debugging-Prozess markiert und diese Markierung während der Generierung beobachtet.

9. Anforderung: Der Generierungsprozess wird dynamisch dargestellt. Die Makros werden durchlaufen, während der generierte Code im Fenster für das Generat nach und nach ergänzt wird. Diesen Fortschritt kann man nach einem Programmstopp anschauen.

   Um den Aufbau einzelner Locations zu verstehen, ist es notwendig, den Generierungsprozess direkt zu beobachten. Hier unterscheidet sich der neue Debugger vom PGD, welcher das Generat nur nach der Generierung anzeigt. Mit einem Debugger, welcher den Generierungsprozess direkt anzeigt, löst sich auch der in Abschnitt 3.3.3 genannte Nachteil des PGD, dass das dynamische Verhalten der Makros nur per Hand Schritt für Schritt nachvollzogen werden kann, auf.

   Konkret soll dies so aussehen, dass man Breakpoints im Programm setzen kann und bei deren Erreichen zeigt das COBOL-Fenster nur die schon generierten Code-Zeilen an.

10. Anforderung: Es können einzelne Schritte rückwärts durchlaufen werden.

    Ein Omniscient Debugger besitzt eine grundsätzlich andere Funktionsweise als ein Breakpoint-Debugger. Um *Anforderung 9* umzusetzen, ist ein Breakpoint-Debugger die

beste Wahl, sodass ein Omniscient Debugger ausgeschlossen werden kann. Wie der GNU Debugger können aber viele Breakpoint-Debugger einzelne Schritte rückwärts ausführen, was auch beim neuen ADS-Debugger umgesetzt werden kann. Um den benötigten Speicherplatz zu reduzieren, ist eine Beschränkung auf wesentliche Rückschritte sinnvoll.

11. Anforderung: Es können Zwischenschritte nach der Abarbeitung einzelner Prozessoren eingesehen werden.

    Die Transient Models stellen Schnappschüsse von bedeutenden Zwischenschritten dar. In der derzeitigen MPS-Implementierung von ADS sind dies zum Beispiel das Anlegen der Locations und deren Befüllung. Es werden jedoch nur sehr große Schritte dargestellt, die Verarbeitung des Java-Quellcodes, der ebenfalls Teil des Generators ist, ist nicht sichtbar. Eine Möglichkeit, diese Technik zu nutzen, ist, die Abarbeitung der einzelnen Prozessoren als Zwischenschritte festzulegen. So kann man dem Nutzer eine Ansicht bieten, die das Programmskelett nach Durchlauf von PROG, dem Programmrahmengenerator, darstellt, oder nach der Abarbeitung aller SPP-Blöcke. Eine Schwierigkeit stellt aber die optische Aufbereitung dieser Informationen dar, da es für diese Zwischenschritte bisher keine Darstellung gibt.

Da mit Blick auf die 9. Anforderung ein Breakpoint-Debugger am geeignetsten ist, werden im Folgenden noch einige Anforderungen formuliert, die aus der Übertragung der Funktionalitäten des GNU Debuggers auf den neuen ADS-Debugger resultieren:

12. Anforderung: Breakpoints können am Rand der Makros gesetzt werden, sodass bei ihrem Erreichen die Generierung gestoppt wird.
13. Anforderung: Variablen können mithilfe von *Watchpoints* betrachtet werden.
14. Anforderung: Die Anweisung *next* geht nach einem Programmstopp in die nächste Anweisungszeile, *step* überspringt Anweisungsblöcke, wie einen `If`-Block und mit *continue* wird der Programmdurchlauf fortgesetzt. Zusätzlich gibt es eine Anweisung *return*, mit welcher man, wie in Anforderung 10 beschrieben, das Programm in großen Schritten rückwärts durchlaufen kann.
15. Anforderung: Eine direkte Auswertung von Ausdrücken, die während der Generierung eingegeben werden, ist nützlich, um auszutesten, wie sich diese auf das Programmverhalten auswirken.

Catchpoints sind auf den ADS-Debugger nur schwer übertragbar, da es keine Exceptions oder vergleichbare Ereignisse gibt. Jedoch ist noch nicht absehbar, ob dies in der modernisierten Version von ADS implementiert wird; in diesem Fall könnte man Catchpoints zur Verfügung stellen.

Das Anzeigen von Variablenwerten wurde in der 6. Anforderung formuliert. Die Übertragung des Aufrufstapels findet sich in der 7. Anforderung.

4.1.6 Anforderungen aus den untersuchten Generatoren

Auf die Generatoren, die ausschließlich Logging-Unterstützung bieten, soll hier nicht näher eingegangen werden. Dieser Aspekt wurde in Abschnitt 4.1.5 schon näher betrachtet.
Doxygen bietet dem Nutzer ein Transient Model an, welches den extrahierten Quellcode mit Markdown und Kommandos zeigt. Da bei ADS nach der Extrahierung der COBOL-Zeilen ebenfalls noch die Verarbeitung der Untergeneratoren stattfindet, lässt sich diese Anforderung übertragen, sie wurde allerdings schon in der 11. Anforderung berücksichtigt.
Das Google Web Toolkit stellt keine Informationen zum Generierungsprozess bereit. Xtext bietet für Xtend ein Matching zwischen Eingabe und Generat, dieses wird durch die vorangehenden Anforderungen jedoch ausreichend ersetzt.
Die Auswertung der Debug-Möglichkeiten von MPS findet sich unter dem Abschnitt zum Thema Transient Models (Abschnitt 4.1.5); die zusätzlichen Informationsquellen werden hier nicht mehr betrachtet, da diese Funktionalitäten als Ersatz eines Breakpoint-Debuggers dienen. Sie werden hier nicht berücksichtigt, da ein Breakpoint-Debugger konstruiert wurde.

4.2 Anpassung der Anforderungen an die Gegebenheiten in MPS

Die Entwicklung des Debuggers erfolgt auf der Grundlage der Generator-Implementierung in MPS. Dadurch ergeben sich einige Vorgaben hinsichtlich der Struktur und des Ablaufes des Debuggers. Aus diesem Grund werden im folgenden Abschnitt die vorher ermittelten Anforderungen an den Debugger geprüft und gegebenenfalls angepasst oder erweitert. Grundsätzlich orientiert sich die Generator-Implementierung in ADS an der originalen ADS-Implementierung, sodass die angepassten Anforderungen größtenteils übertragbar sind. Details sind jedoch noch nicht bekannt und müssen bei der Implementierung bedacht werden.

4.2.1 Zeitlicher Ablauf der Generierung mit MPS

Der zeitliche Ablauf der Generierung kann in folgende Schritte unterteilt werden:

1) Ausführung des *ADS-Interpreters*
2) Aufruf von Templates
3) Ausführung der *Weaving-Rule*
4) Textgenerierung

1) Ausführung des *ADS-Interpreters*

Der ADS-Interpreter ist eine Java-Klasse, welche für jede ADS-Anweisung eine oder mehrere Funktionen zur Verarbeitung anbietet. Diese Funktionen werten die Ausdrücke aus und

speichern Variablenwerte und verschiedene Zustände. Außerdem werden COBOL-Ausdrücke verarbeitet. Einen Unterschied zur Behandlung der übrigen Anweisungen gibt es bei SPP- und PSD-Blöcken. Bei diesen werden nur die Makroanweisungen ausgewertet, die SPP- und PSD-Anweisungen werden unverarbeitet weitergeleitet und erst in Schritt 4 ausgewertet. In Abbildung 7 sieht man einen Ausschnitt aus der Klasse *ADSInterpreter*. Es ist allerdings keine Ansicht nach Ausführung dieses Schrittes verfügbar.

```
public class ADSInterpreter{
 private gencontext genContext;
 private map<string, string> localVariables = new hashmap<string, string>;
 private map<string, string> globaleVariables = new hashmap<string, string>;
 private map<string, string> currentStaticParameters;

 public enum locationStatusType {
   KNOWNBUTUNUSED()
   KNOWNANDUSED()
   UNKNOWN()

 }
```

Abbildung 7: Ausschnitt aus der Klasse ADSInterpreter

2) Aufruf von Templates

Im zweiten Schritt wird zunächst ein übergeordnetes Programmskelett angelegt, welches `LocationTree` heißt und wiederum einige `LocationTrees` enthält. Nun werden Knoten vom Typ `NewLocation` und `COBOL` transformiert und in ein Template eingebettet, damit sie in diesen `LocationTree` eingefügt werden können. In Abbildung 8 sind die Transformationen eines `COBOL`-Knotens mithilfe eines Templates sowie der gefüllte `LocationTree` dargestellt.

Abbildung 8: Transformation der Knoten und Aufbau des LocationTrees

3) Ausführung der *Weaving-Rule*

Für die `Do`-Schleife des Prozessors SPP gibt es noch eine Sondertransformation. Übertragen in den COBOL-Code bedarf sie einer zusätzlichen Hilfsvariable, die im Deklarationsblock deklariert werden muss. Um dort Knoten nachträglich einfügen zu können, gibt es eine sogenannte *Weaving-Rule*, die bei ihrer Anwendung einen zusätzlichen Knoten in den LocationTree einsetzt. Abbildung 9 zeigt beispielhaft zwei in das COBOL-Programm eingefügte Knoten.

```
LocationTree: $WORKV
Content:  "     10 DX-SPP-1 PIC S9(9) COMP-5."
Content:  "     10 DX-SPP-2 PIC S9(9) COMP-5."
Content:  "     10 FILLER     PIC S9(8) COMP SYNC RIGHT."
Content:  "* ----------------------------------------------> WORK01 USER"
```

Abbildung 9: Zwei durch die Weaving-Rule eingefügte Knoten

4) Textgenerierung

Zuletzt wird für die COBOL-Knoten und die Prozessoren Text an die entsprechenden Stellen im LocationTree generiert. An dieser Stelle erfolgt erst die Auswertung der PSD- und SPP-Blöcke, wie in Schritt 1 beschrieben. Abbildung 10 zeigt einen Ausschnitt des in diesem Schritt generierten COBOL-Programms.

```
000001*PROGMODE: BATCH,SUBPROGRAM
000002 IDENTIFICATION DIVISION.
000003**PDL*201509221357/N02353V00/03/PROGRAM N02353V00 ADS-50
000004*
000005*                          PROGRAM GENERATED BY ADS REL 50
000006*
000007 PROGRAM-ID.                N02353V00.
000008 AUTHOR.                    NOAUTHOR.
000009 DATE-WRITTEN.              22 09 2015.
000010 ENVIRONMENT DIVISION.
000011 CONFIGURATION SECTION.
000012 SOURCE-COMPUTER.           MICROFOCUS.
000013 OBJECT-COMPUTER.           MICROFOCUS.
000014 DATA DIVISION.
000015 WORKING-STORAGE SECTION.
000016* ------------------------------------------------> WORK77 USER
```

Abbildung 10: Ausschnitt aus einem generierten COBOL-Programm

Der Programmrahmengenerator PROG wird in MPS durch den Aufruf eines Untermakros nachgebildet. Dieses Makro enthält die notwendigen Anweisungen und logischen Bedingungen, um den Programmrahmen zu erstellen.

4.2.2 Anpassung und Konkretisierung der Anforderungen

Die Anforderungen 1 bis 4 sind allgemeine Anforderungen, die keiner Anpassung an den Generator in MPS bedürfen.

Um die Zusammenschau einander entsprechender Codezeilen aus der 5. Anforderung bereitzustellen, kann man eine AST-ID nutzen. In der MPS-Implementierung des Generators besitzt jede Anweisung eine solche AST-ID, welche für den Debugger genutzt werden kann.

Die 6. Anforderung lässt sich mithilfe des ADS-Interpreters gut umsetzen, da hier alle aktiven Variablen gespeichert sind. Hierbei muss man jedoch beachten, dass zu jedem Wert einer Variable auch das gerade aktive Makro und die Zeile gespeichert werden, damit auch im Nachhinein nachvollzogen werden kann, wo genau die Variable diesen Wert angenommen hat.

Für Anforderung 7, den Aufrufgraphen, könnte man eine XML-Datei erstellen, wie es auch aktuell gehandhabt wird. Aus dieser wird beim Debuggen ein Graph erstellt. Als Ergänzung muss hier noch die Codezeile gespeichert werden, in welcher der Aufruf geschieht, um eine Verlinkung herzustellen.

Die Markierung von COBOL-Zeilen aus der 8. Anforderung kann ebenfalls über die AST-ID geschehen, jedoch kann die Markierung nur so genau sein, wie die AST-ID einer Anweisung direkt zugeordnet werden kann. Dies stellt jedoch keine Einschränkung dar, da die AST-IDs für sehr kleine Abschnitte im Code vergeben sind.

Anforderung 9 stellt die größte Herausforderung in der Implementierung mit MPS dar.

Da die Generierung aktuell in mehreren Schritten geschieht, wie in Abschnitt 4.2.1 beschrieben, wird zum Zeitpunkt, in dem der ADS-Interpreter läuft, noch keine COBOL-Anweisung generiert. Dies geschieht erst in Schritt 4. Dadurch kann mit der aktuellen Implementierung die Generierung einzelner Zeilen nicht zeitgleich mit dem Durchlaufen der Makros abgebildet werden. Deshalb werden an dieser Stelle drei Möglichkeiten aufgezeigt, wie dieser Konflikt aufgelöst werden kann:

1) Man verzichtet auf die COBOL-Ansicht während der Generierung und kann den generierten Code nur im Nachhinein als Ganzes betrachten.
   Vorteil: Die derzeitige Funktionsweise des Generierungsprozesses kann beibehalten werden und es müssen während der Generierung keine zusätzlichen Informationen gespeichert werden wie bei Lösung 3.
   Nachteil: Dies entspricht weniger der Funktionsweise herkömmlicher Breakpoint-Debugger, bei denen das Programm während des Debuggens schon ausgeführt wird. Außerdem geht damit die Funktionalität verloren, die eine Verbesserung gegenüber dem ADS-PGD darstellt und dessen unzureichende Illustrierung des Programmablaufes verbessern soll.
2) Die Generierungsreihenfolge wird angepasst. Hierzu wird zunächst das Makro durchlaufen, um den `LocationTree` aufzubauen. Nach dem ersten Schritt ist also die grobe Programmstruktur schon vorhanden. In einem zweiten Schritt werden dann die Makros mitsamt den Unterprozessoren verarbeitet und COBOL-Anweisungen werden direkt in das schon aufgebaute Programmgerüst generiert.
   Vorteil: Die Erstellung der groben Programmstruktur mit den Locations ist ein klar abgetrennter Schritt, was zunächst eine sehr übersichtliche Ansicht für den Nutzer bietet. Hiermit kann man Anforderung 9 effizient (bezüglich benötigtem Speicherplatz und Ausführungszeit, verglichen mit Lösung 3) umsetzen.
   Nachteil: Die Abarbeitung des Programms geschieht nicht mehr nach der dargestellten Reihenfolge. Dies ist nicht intuitiv. Außerdem muss die derzeitige Implementierung sehr aufwändig verändert werden.
3) Die Generierung der COBOL-Anweisungen wird nur simuliert und entspricht nicht dem realen Programmablauf. Hierzu werden in einem ersten Durchlauf Informationen über das Programm gesammelt und in einer XML-Datenstruktur gespeichert. Anschließend werden der Debugger ausgeführt und die Makros durchlaufen. Während dieser Phase kann der Nutzer mit dem Debugger interagieren und die Generierung wird in dem COBOL-Fenster dargestellt. Das Sammeln und Speichern der

notwendigen Informationen zum Debuggen findet dabei nur auf Wunsch des Nutzers statt.

Dieses Verhalten entspricht der Funktionsweise des ADS PGD, der ebenfalls während der Generierung Informationen sammelt und diese in XML-Dateien speichert. Anschließend kann man die Informationen in einer gesonderten Ansicht betrachten. Unterschiede bestehen im automatischen Durchlaufen der Programme und im dynamischen Anzeigen der Generierung durch den neuen Debugger.

Vorteil: Die Implementierung des Debuggers kann beibehalten werden. Gegebenenfalls können schon bestehende XML-Strukturen genutzt und erweitert werden. Außerdem ist eine Debug-Sitzung mithilfe der gespeicherten Informationen beliebig oft aufrufbar und auch die Implementierung von Rückschritten in der Programmausführung ist auf einfache Weise möglich.

Nachteil: Es müssen sehr viele Informationen gesammelt und gespeichert werden, was bezüglich der Speichernutzung ineffizient ist.

Die 10. Anforderung, einzelne Schritte rückwärts zu durchlaufen, wäre mit Lösung 3 vergleichsweise einfach realisierbar. Bei den ersten beiden Lösungen müssten zusätzliche Informationen über vorherige Zustände des Programms gespeichert werden. Hier wäre es einfacher, die Rückschritte auf gewisse Zustände zu begrenzen, beispielsweise auf den Aufruf eines Untermakros.

Anforderung 11, die Darstellung von Zwischenschritten nach der Abarbeitung von Unterprozessoren, stellt nur für PROG eine sinnvolle Anforderung dar. PROG ist in MPS als Untermakro implementiert, das, wenn es genutzt wird, als erste Anweisung in einem Programm aufgerufen wird. Dadurch stellt es tatsächlich einen ersten, klar abgetrennten Schritt dar, den man dem Nutzer zeigen könnte. Allerdings wird dieser Schritt im Moment nicht gespeichert. Mit Lösung 3 wäre dies ohne viel zusätzlichen Aufwand möglich, bei Lösung 1 und 2 müssten zusätzlich Speicherfunktionalitäten für diese Ansicht angelegt werden.

Die Darstellung des Generierungsstandes nach der Verarbeitung von anderen Unterprozessoren ist bei der aktuellen Funktionsweise nicht sinnvoll. Aktuell werden diese erst im letzten Schritt verarbeitet, die Ansicht enthielte genau das Gegenteil der tatsächlichen Verarbeitungsreihenfolge, wenn man einen Zwischenstand nach der Auswertung der Prozessoren aber ohne die verarbeiteten ADS-Anweisungen anzeigen würde.

Das Setzen von Breakpoints in Anforderung 12 wirft die Frage auf, ob dies in allen Generatoren gleichermaßen möglich ist, also auch in den Untergeneratoren. Da PROG wie die ADS-Anweisungen in anderen Makros vom ADS-Interpreter verarbeitet wird, soll man hier

auch Breakpoints setzen können, um dem Nutzer Einheitlichkeit zu gewährleisten. Lässt man jedoch Breakpoints in anderen Prozessoren zu, wie SPP und PSD, wirft das die Frage auf, wie die Verarbeitungsreihenfolge dargestellt wird.
Bei Lösung 1 würde man bei der Trennung von ADS-Anweisung und der Verarbeitung der Unterprozessoren bleiben. Da es aber hier keine Darstellung des aktuellen Generierungsstandes gibt, wären Breakpoints sowieso nicht nötig. Aus diesem Grund wird Lösung 1 ausgeschlossen. Bei Beibehaltung der aktuellen Implementierung schränkt sie die formulierten Anforderungen zu stark ein.
Setzt man Lösung 2 um, kann man die Verarbeitung der Prozessoren im gleichen Schritt erfolgen lassen wie die Verarbeitung der ADS-Anweisungen. Das hat den Vorteil, dass die Makros ohne Sprünge innerhalb eines Programms verarbeitet werden, was für den Nutzer intuitiver ist. Bei dieser Variante kann man Breakpoints auch bei den Prozessoren setzen, da diese in jeder Hinsicht gleichwertig zu den ADS-Anweisungen behandelt werden.
Lösung 3 lässt offen, in welcher Reihenfolge die Generierung dargestellt wird. Es erscheint jedoch sinnvoll, bei beiden Varianten Breakpoints an jeder Stelle des Programms zuzulassen, auch der Code für die Prozessoren kann lang und schlecht verständlich sein. Behält man die nachgeordnete Verarbeitung der Prozessoren bei, bekommt der Nutzer angezeigt, zu welchem Zeitpunkt die Verarbeitung der Prozessoren einsetzt, damit er eventuelle Rücksprünge im Programmablauf nachvollziehen kann.
Anforderung 13, die Implementierung von Watchpoints für Variablen, dürfte bei Lösung 2 und 3 ohne viel Aufwand möglich sein.
Zu Anforderung 14 gibt es keine weiteren Anpassungen. Die *return*-Anweisung ist im Abschnitt zu Anforderung 10 schon behandelt worden.
Wie die direkte Auswertung von Ausdrücken während des Debuggens aussieht, wird in dieser Arbeit nicht näher diskutiert.

4.3 Verallgemeinerung der Ergebnisse auf Generatoren

Wie Rosenberg in seinem vierten Prinzip (*Debugging Trails System Development)* betont, hinkt die Entwicklung von Debuggern oftmals den existierenden Programmen hinterher. Dies hat die Untersuchung in dieser Arbeit ebenfalls bestätigt, auch wenn diese nicht repräsentativ war. Es gibt hier noch viel Potenzial, um die Arbeit mit den Generatoren, insbesondere das Auffinden von Fehlern, zu vereinfachen.
In seiner direkten Übernahme der COBOL-Statements in eine neue Datei ähnelt ADS am meisten den Generatoren des Typs Code Munger. Um diesen recht einfachen Zusammenhang darzustellen, wurde in dieser Arbeit das Fenster mit den erzeugten COBOL-Statements

gefordert, in dem man während des Debugger-Laufes die Generierung beobachten kann. Diese Gegenüberstellung bietet sich bei Generatoren dieses Typs an. Für vergleichsweise schlichte Generatoren, wie die betrachteten Dokumentationswerkzeuge, die Text anders formatiert übernehmen, kann diese einfache Darstellung schon reichen.

Bei komplexeren Generatoren, wie den Tier-Generatoren ist ein aufwändigerer Debugger nötig. Wie in Abschnitt 2.1 beschrieben, führen horizontale Generatoren durch verschachtelte Transformationen, brechen Strukturen auf und erstellen neue Gliederungseinheiten. Hier wäre eine grafische Übersicht hilfreich, möglicherweise ähnlich zu den Ansichten des Whyline-Debuggers. Außerdem kann ein Vergleich von übereinstimmenden Zeilen dem Nutzer des Debuggers helfen, die Herkunft einer Zeile zu ermitteln und so Zusammenhänge zu erkennen. Auch die dynamische Darstellung des Generierungsprozesses ist eine Möglichkeit, das Verhalten des Generators zu illustrieren, wie in Abschnitt 2.2.1 beschrieben. Vor allem, wenn verschiedene Eingaben verarbeitet werden, ist es wichtig, ihr Zusammenspiel zu verstehen.

Zur Betrachtung von Templates vor ihrer Befüllung bieten sich Ansichten analog zu den Transient Models an. Auch das Template ist als Eingabe für das Verhalten des Generators relevant und soll deshalb für den Nutzer zugänglich sein.

Generatoren, die als Eingabe Turing-vollständige Sprachen bekommen (von Herrington als *Full Domain Languages* bezeichnet (vgl. [Herrington 2003, 33f])), ähneln in ihrer Eingabe gewöhnlichen Turing-vollständigen Programmiersprachen. Bei Letzteren werden in der Regel Breakpoint-Debugger eingesetzt. Deshalb ist es für einen Nutzer vertraut, wenn auch bei ähnlichen Generatoren Breakpoint-Debugger angeboten werden. Allerdings ist es noch nicht erforscht, inwieweit andere Arten von Debuggern für Generatoren des Typs Full Domain Language nützlich sein können, beispielsweise Debugger vom Typ Query-based-Debugger.

Bei einer Generierung in mehreren Phasen, können Transient Models hilfreiche Informationen liefern. Hiermit lassen sich die einzelnen Phasen übersichtlich darstellen und Zwischenergebnisse überprüfen. Allerdings muss der Nutzer auch verstehen, welche Eingabe, in welchem Schritt wie verarbeitet wird, sonst lässt sich das Verhalten mithilfe der Transient Models nur schwer nachvollziehen. Hier ist das Generation Trace Tool von JetBrains MPS beispielhaft zu nennen, mit welchem die einzelnen Transformationsschritte, die auf ein Codefragment angewendet wurden, übersichtlich dargestellt werden.

Insgesamt lässt sich feststellen, dass, wie in Kapitel 4.2.2 beschrieben, ein Debugger für einen Generator sehr genau an diesen angepasst werden muss. Aufgrund des komplexen Zusammenspiels der einzelnen Generierungsschritte und der aufwändigen Implementierung

in MPS können einige theoretische Anforderungen nur mit Umstrukturierungen des Generators oder mit anderen sehr aufwändigen Strukturen umgesetzt werden. Dies ist ein Nachteil bei der Debugger-Entwicklung für Generatoren. Alternativ ist es sinnvoll, den Debugger parallel zum Generator zu entwerfen und zu entwickeln, um solche Probleme frühzeitig zu umgehen.

5 Fazit

5.1 Zusammenfassung

Das Ziel dieser Arbeit war die Ermittlung von Anforderungen für den Debugger eines Generators. Zu diesem Zweck wurden zunächst Debuggertypen identifiziert, es konnten fünf relevante Typen gefunden werden. Nachdem diese beispielhaft skizziert wurden, folgte eine Klassifizierung von Generatoren. Diese wurde dem Buch von Rosenberg [1996] entnommen, zusammen mit Beispielen zu den betreffenden Generatortypen. Hieraus ergab sich eine Liste von insgesamt 21 zu untersuchenden Generatoren zuzüglich des in dieser Arbeit betrachteten Generators ADS. Nachdem die Dokumentation der Generatoren ausführlich studiert wurde, gab es folgende Ergebnisse: Bei sechs Generatoren konnten keine aussagekräftigen Quellen gefunden werden. Fünf Generatoren besitzen keinerlei Debugging-Unterstützung für die Generierung, zwei der Beispiele verwenden Transient Models. ADS hat einen Debugger, der einzigartig unter den betrachteten Generatoren ist und die restlichen stellen Logging-Informationen zur Verfügung.

Auf Basis dieser Ergebnisse konnten anschließend 15 Anforderungen an den neuen ADS-Debugger formuliert und diskutiert werden. Als mögliche Lösung wurde ein Breakpoint-Debugger konzipiert, der in seinem Aufbau dem ADS-Post-Generation-Debugger ähnelt und zusätzlich eine Ansicht des Programm-Templates bietet. Anschließend folgten einige Überlegungen zur praktischen Implementierung des Debuggers auf der Basis von JetBrains MPS. Hier bereitete vor allem die Anforderung der dynamischen Anzeige der generierten COBOL-Zeilen Probleme, es konnten aber zwei mögliche Lösungsansätze ermittelt werden.

Zuletzt folgte eine Verallgemeinerung der Ergebnisse auf Generatoren. In diesem Abschnitt ist noch einmal hervorgehoben worden, dass ein Debugger für einen Generator vor allem genau an den Generatortyp, seine Eingabe und Funktionsweise angepasst werden muss. Ebenfalls zu berücksichtigen ist die Komplexität der Generatortransformationen. Dieser entsprechend muss die Komplexität des Debuggers steigen, um eine angemessene Zahl und Komplexität an Informationen darstellen zu können.

5.2 Kritik

Das in dieser Arbeit zitierte Buch von Rosenberg stammt aus dem Jahr 1996. Seit dieser Zeit ist die Debugger-Entwicklung fortgeschritten, es entspricht möglicherweise in manchen Punkten nicht mehr dem aktuellen Stand der Entwicklung. Allerdings sind die Grundlagen heute immer noch gültig, weshalb die Entscheidung für dieses Buch fiel. Außerdem stellt es ein Standardwerk für die Debugger-Entwicklung dar.

Ein weiterer kritischer Punkt ist das Buch *Code Generation in Action* [Herrington 2003]. Dieses ist in großen Teilen eher für den praktischen Anwender geschrieben. Außerdem bildet die Klassifizierung der Generatoren keine disjunkten Mengen, was die Klassifizierung von ADS zeigt. ADS ließ sich nicht eindeutig nur einem Generatortyp zuordnen, was möglicherweise die Untersuchungsergebnisse beeinflusste.
Außerdem ist bezüglich der Untersuchung anzumerken, dass nur eine kleine Auswahl von Generatoren betrachtet wurde. Diese sind nicht repräsentativ ausgewählt. Damit stellen sie möglicherweise nicht die gesamte Bandbreite der verfügbaren Debugger dar, was das Untersuchungsergebnis beeinträchtigt.
Zuletzt ist noch anzumerken, dass bei der Anforderungsermittlung der ADS-PGD teilweise als Grundlage diente. Durch das Vorhandensein des PGD waren einige Anforderungen schon gegeben und es gab einige Merkmale, die als nützlich gewertet und deshalb beibehalten worden sind, was vor allem die Auswahl der verschiedenen Ansichten beeinflusst hat.

5.3 Offene Fragen

Die wichtigste Frage, die hier nicht beantwortet werden konnte, ist, ob die praktische Umsetzung des Debuggers wie beschrieben funktioniert.
Außerdem ist es wichtig, den Nutzen des in dieser Arbeit beschriebenen Debuggers zu zeigen. Ziel der Arbeit war es, dem Entwickler alle notwendigen Informationen zu geben, damit dieser Fehler möglichst effizient auffindet und das Verhalten des Generators besser verstehen kann. Diese Ziele wurden in Kapitel 1.1 formuliert. Aufgrund des begrenzten Umfangs der Arbeit konnten allerdings weder die Implementierung noch die Evaluation der Anforderungen umgesetzt werden.
Eine weitere offene Frage, die sich im Laufe der Arbeit ergeben hat, ist, wie ein Query-based-Debugger für einen Generator aussehen muss. In Kapitel 4.1.5 wurden Fragen für einen solchen Debugger grob angerissen. Eine ausführlichere Untersuchung müsste aber ermitteln, welche Fragen für einen Entwickler wirklich relevant sind und wie die Informationen, die als Antwort zurückgeliefert werden, bestmöglich dargestellt werden können. Außerdem stellt sich die Frage, ob Debugger dieses Typs für Generatoren nützlich und passend sind.

Literaturverzeichnis

[Alshannikov 2013] Alshannikov, I.; Pech, V., Generator User Guide Demo6,2013, URL: https://confluence.jetbrains.com/display/MPSD33/Generator+User+Guide+Demo6#GeneratorUserGuideDemo6-generationtracertool, gelesen am 11.03.2016.

[AndroMDA 2014] o.V., AndroMDA Configuration, The AndroMDA Team, 2014, URL: http://andromda.sourceforge.net/configuration.html, gelesen am 11.03.2016.

[Bettini 2013] Bettini, Lorenzo, Implementing domain-specific languages with Xtext and Xtend, Packt Publishing, Birmingham, 2013.

[CodeCharge 2012] o.V., CodeCharge Studio 5 Full On-line Documentation, YesSoftware Inc., 2012, URL: http://docs.codecharge.com/studio2/html/index.html?http://docs.codecharge.com/studio2/html/ProgrammingTechniques/HowTo/Debugging/DisplayingOutput.html, gelesen am 10.03.2016.

[Corley 2013] Corley, J., Debugging for Model Transformations, in: Martin Gogolla (Hrsg.), Proceedings of the MODELS 2013 Doctoral Symposium co-located with the 16th International ACM/IEEE Conference on Model Driven Engineering Languages and Systems (MODELS 2013), CEUR-WS.org, Miami, USA, 2013, S. 17-24.

[Czarnecki/Eisenecker 2005] Czarnecki, K., Eisenecker, U. W., Generative Programming. Methods, Tools and Applications, Addison Wesley, Boston, 6. Aufl., 2005.

[Delta Software Technology GmbH 2012] Delta Software Technology GmbH, ADS™ Band 1: Grundlagen, 2012.

[Denker et al. 2006] Denker, M., Ducasse, S., Hofer, C., Design and Implementation of a Backward-In-Time Debugger, in: Lecture Notes Informatics, Proceedings of NODe 2006, 2006, S. 17-32.

[Efftinge 2016] Efftinge, S., Xtext Reference Documentation. Configuration, 2016, URL: https://eclipse.org/Xtext/documentation/302_configuration.html#logging, gelesen am 11.03.2016.

[EJBGen Reference 2016] o.V., EJBGen Reference, BEA Systems, 2016, URL: https://docs.oracle.com/cd/E13222_01/wls/docs81/ejb/EJBGen_reference.html, gelesen am 10.03.2016.

[GWT 2014] o.V., Build a GWT App. Debugging, Google, 2014, URL: http://www.gwtproject.org/doc/latest/tutorial/debug.html, gelesen am 14.03.2016.

[Herrington 2003] Herrington, J., Code generation in action, Manning, Greenwich Conn., 2003.

[Iron Speed 2015] o.V., Iron Speed. Tracing and Event Logging, 2015, URL: http://www.ironspeed.com/Designer/12.2.0/Web-Help/desktop/Part_VIII/Tracing_and_Event_Logging.htm, gelesen am 10.03.2016.

[JavaDoc 2010] o.V., JavaDoc Reference Guide, Oracle, 2010, URL: http://docs.oracle.com/javase/1.5.0/docs/tooldocs/solaris/javadoc.html, gelesen am 10.03.2016.

[Kehrer et al. 2013] Kehrer, T., Kolb, B., Pavletic, D., Raza, S.A., Voelter, M., Extensible Debuggers for Extensible Languages, in: Softwaretechnik-Trends 33 (2013) 2, S. 51-52.

[Ko/Myers 2008] Ko, A.J., Myers, B.A., Debugging reinvented, in: Schäfer, W., Dwyer, M.B., Gruhn, V. (Hrsg.), Proceedings of the 30th international conference on Software engineering, 2008, S. 301-310.

[Pesch et al. 2015] Pesch, R., Shebs, S., Stallman, R., Debugging with GDB. The GNU Source-Level Debugger, 10.Aufl., 2015.

[phpDocumentor] o.V., phpDocumentor Documentation, URL: https://www.phpdoc.org/docs/latest/references/configuration.html, gelesen am 10.03.2016.

[RDoc 2005] o.V., RDoc, Rubyforen, 2005, URL: http://wiki.ruby-portal.de/RDoc, gelesen am 10.03.2016.

[Rosenberg 1996] Rosenberg, J.B., How debuggers work. Algorithms, data structures and architecture, Wiley, New York, 1996.

[Seefeld 2004] Seefeld, S., Synopsis Reference Manual, 2004, URL: http://synopsis.fresco.org/docs, gelesen am 10.03.2016.

[Talin 2000] Talin, Scandoc, 2000, URL: http://scandoc.sourceforge.net/scandoc.html, gelesen am 10.03.2016.

[van Engelen 2016] van Engelen, R., gSOAP 2.8.29 User Guide, Genivia Inc., 2016.

[van Heesch 2015] van Heesch, Dimitri, Doxygen Manual, 2015, URL: http://www.stack.nl/~dimitri/doxygen/manual/markdown.html#markdown_debug, gelesen am 10.03.2016.

[Wikipedia 2016] Wikipedia, Grace Hopper, 2016, URL: https://de.wikipedia.org/wiki/Grace_Hopper, gelesen am 01.04.2016.

[Zeller 2009] Zeller, A., Why programs fail. A guide to systematic debugging, Morgan Kaufmann, Elsevier science distributor, San Francisco, Calif., Oxford, 2.Aufl., 2009.

Ehrenwörtliche Erklärung

Ich versichere, dass ich die Bachelorarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Darüber hinaus versichere ich, dass die elektronische Version der Bachelorarbeit mit der gedruckten Version übereinstimmt.

_____ _____

Ort, Datum Unterschrift